TIBCO Business Studio™ Forms User's Guide

Software Release 4.3
April 2019

Important Information

Contents

Figures

TIBCO Documentation and Support Services

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit https://docs.tibco.com.

Product-Specific Documentation

Documentation for TIBCO products is not bundled with the software. Instead, it is available on the TIBCO Documentation site. To directly access documentation for this product, double-click the following file:

TIBCO_HOME/release_notes/TIB_business-studio-bpm-cloud-edition_*version*_docinfo.html

The following documents for this product can be found on the TIBCO Documentation site:

- TIBCO Business Studio™ Concepts
- TIBCO Business Studio™ Modeling User's Guide
- TIBCO Business Studio™ - Analyst Edition User's Guide
- TIBCO Business Studio™ - BPM Implementation
- TIBCO Business Studio™ Forms User's Guide
- TIBCO Business Studio™ Simulation User's Guide
- TIBCO Business Studio™ Customization
- TIBCO Business Studio™ - Analyst Edition Installation
- TIBCO Business Studio™ - BPM Edition Installation
- TIBCO Business Studio™ iProcess to BPM Conversion

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit http://www.tibco.com/services/support.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at https://support.tibco.com.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to https://support.tibco.com. If you do not have a user name, you can request one by clicking Register on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to https://community.tibco.com.

Concepts

It is important to understand concepts and terminology related to creating forms in TIBCO Business Studio.

The Modeling Environment for Forms

The context for creating and deploying forms is the BPM Modeling perspective of TIBCO Business Studio.

An understanding of the terms and concepts explained in the TIBCO Business Studio guides and tutorials on Process Modeling and the Business Object Modeler are useful for performing the procedures used to create and deploy forms. In addition, familiarity with the basics of the Eclipse environment makes it easier to work with TIBCO Business Studio and Forms. You can refer to The Workbench to get a general idea about the Eclipse workbench. You can also see the Concepts chapter in the *Workbench User Guide* for information about projects, folders, perspectives, views, menus, and toolbars as they are applied in Eclipse. That guide, as well as all guides related to TIBCO Business Studio and your Eclipse environment, can be accessed by clicking **Help Contents** on the **Help** menu.

The Eclipse *Workbench User Guide* describes the ways you can customize your Eclipse environment to suit your personal preferences.

The Form and Form Elements

Forms can be created as a stand-alone resources. A form is a model of a user interface designed for a particular task or type of task. When deployed to an execution environment, the form drives the user interface or interaction with the human who has been assigned the associated task.

The user interface helps the user to complete the task quickly and correctly by presenting information that is relevant to the task, asking for information that is required, and validating any information that the user provides. All of these capabilities are modeled within the form in TIBCO Business Studio.

Forms contain user interface controls and panes as well as input and output parameters. They may also contain control validations, bindings, actions, and rules.

A form contains two main types of objects in its visual layout: panes and controls. Each pane and control on a form has a **Properties** view associated with it, where you can view and edit the properties that determine the layout and functionality of that object. See the figure Form Elements for the form element presentation.

Form Elements

Panes

Panes are used as a mechanism to control the layout of the form.

Several types of panes are found in the palette. Vertical and horizontal panes support the visual alignment of controls as well as other nested panes. When nested inside a special tabbed pane, these panes behave similar to tab pages. Error messages from control validations are displayed in a Message pane.

Panes can also specify the default rendering of controls they contain (called child controls), such as the font and label position.

Panes and controls may be generated automatically from an underlying Business Object Model (BOM) or an application-specific model in a product making use of TIBCO Business Studio Forms.

Panes and controls can be added manually by clicking the icon for the desired object in the palette and then clicking again in the location where you wish to place the object on the canvas. The object can also be inserted by clicking the item in the palette and dragging it to the desired location in the canvas.

Ergonomic best practice is to use the "click-move-click" gesture instead of "click-drag-drop" in order to avoid strain on the Carpel Tunnel that can cause Repetitive Strain Injury (RSI).

See Panes for more details.

Controls

Controls are user input elements. They include text controls, date and time controls, radio buttons, check boxes, and images.

They enable the display and capture of data in different ways. Controls have text labels, and usually have fields that display and accept input from a user. A number of settings can be configured for a control, such as labels, hints, visibility, fonts, and others. Control labels, hints, and choice labels can be localized in properties files.

See Controls for more details.

Parameters

Parameters represent the data passed between the form and the containing application. The values of parameters can be bound to the values of controls, or to the other settings on controls and panes.

Output parameters can also be mapped to controls. The parameter can be an IN parameter, which means the value is read-only and provided to the form when it is opened. An OUT parameter is provided by the user and sent back to the containing application. A parameter can also be IN/OUT.

Parameters have unique names within a specific form. Each parameter has a type, which can either be one of the pre-defined primitive types such as Integer or DateTime, or a complex type defined by the user in a Business Object Model.

For more details, see Configuration of Parameters.

Validations

Validations are used to check the validity of data specified by the user and specify an appropriate message to display to the user in the event the validation fails. Validations are executed either when the form is submitted or when the value of the control is updated.

Errors and warnings that result from validation are displayed in the **Problems** view. Validation messages can be localized.

See Form Builders and Form Validation for more details.

Bindings

Bindings are used to synchronize values within a form, such as binding the value of a parameter to the value of a control, or using the value of one control to update the visible flag on another control or pane.

See Bindings for more details.

Actions

An action is a unit of executable functionality. Actions have names and can be executed from rules or scripts. Predefined system actions include submit, reset, and validate.

See Actions for more details.

Rules

Rules are used to encapsulate business logic that is to be executed at certain points within the form. A rule specifies one or more actions that are to be executed in response to one or more event triggers within the form.

See Rules for more details.

Form Builders and Form Validation

The Form Builders and Validation Builder are Eclipse builders that perform various post-processing operations on a form model when the project is built. Generally speaking, the Eclipse auto-build feature will be enabled, which causes an *incremental build* to run automatically whenever a file is saved. When you create a new Business Studio project that includes forms functionality, the New Project Wizard configures the project with the Form Builders and Validation Builder.

Consult the *Eclipse documentation* for further information on the Eclipse build system.

The Validation Builder also performs live validation, which occurs automatically whenever any aspect of the form is modified through the Form Designer canvas, **Outline** view, or **Property** view. Form validations can be configured via the Preferences dialog at **Window** > **Preferences** > **Form Designer** > **Errors/Warnings** .

For each of the validation rules enforced by the Validation Builder, you can use the dropdown list to configure the severity of each problem as **Error, Warning, Info,** or **Ignore**.

The default problem severities are carefully chosen to minimize the possibility of errors at runtime. Change them only on the recommendation of TIBCO Support.

Form Builders

The Form Builders externalize display strings from the form model into property resource files with the path name /<project>/<form-folder>/<form-name>.properties, where

<project> is the project name,

<form-folder> is the folder containing the form file, and

<form-name> is the unqualified name of the form file, minus the .form file extension.

To create a localized version of a form, you will make a copy of this .properties file, rename it by appending the appropriate standard two-character ISO language code (and, optionally, country and variant codes), and translate the strings into the desired language.

For more information about how to localize a form, see The Form and Form Elements .

Validation Builder

The Validation Builder performs these functions:

- Analyses the form model for general syntactical and semantic errors and inconsistencies
- Applies constraints specific to the target platform/version
- Reports any such problems as *problem markers*, which show up in the **Problems** view. To make it easier to locate problems, the problem markers for errors also appear as decorator icons adjacent to the offending form element in the Project Explorer, the **Outline** view, and in the Form Designer. For more information about problem markers, see Problem Markers.

Viewing the Build Configuration of a Project

You can see the builders for a particular project in the project properties.

Procedure

1. In the Project Explorer view, right-click a TIBCO Forms project to display the **Context Menu** and click **Properties**.
2. Click **Builders** in the left-hand panel.

   The entries for Validation Builder and Form Builders are displayed in the right-hand panel.

Data Mappings File

The data mappings file provides mapping information between the form-level parameters and the controls or panes that render or update those parameters. The generation of this file is optional.

You can use the information in the file to determine which controls and panes of a form are bound to the form-level parameters and the properties within those parameters. For each bound pane and control, a key is generated that represents a full path to the property to which the control or pane is bound.

Any control or pane whose value is directly or indirectly bound to a form-level parameter, is listed in the data mappings file. If a control is bound to a pane value or a data field value, and if that value is bound to a parameter, the key is expressed in terms of the full path from the underlying parameter. The generated key:value pairs within the data mappings file are accessible from form scripts.

The data mappings file is automatically added to the form as an external resource and the key:value pairs within it are accessible in the same manner as other property files associated with the form. It is not generated for default forms.

A single mappings file covers all the control and pane bindings in the form, even those that occur within nested embedded forms. The control and pane names within the mappings file correspond to the names of the embedded controls and panes after they are prefixed. The generated keys remain stable as long as the name of the underlying parameter and those of the properties on the path remain the same.

Structure of the Data Mappings File

The Form Designer generates the data mappings file in the Presentation Resources folder for each form.

The naming convention is as follows:

```
<form-name>.mappings.properties
```

The mappings file provides keys that correspond to the form parameters and provide a reference to the control or pane that renders the value. The value of any given key depends only on the name of the parameter upon which it is based. As long as the parameter names remain stable, the generated keys also remain stable.

The format of each line in the property file is as follows:

```
<key>=<value>
where:
  <key> ::= <bindable-name> ['$' <property-name>]*
  <bindable-name> ::= 'param'_<element-name>
  <element-name> ::= [a-zA-Z][0-9a-zA-Z_]*
  <property-name> ::= [a-zA-Z_][a-zA-Z0-9_]*
  <value> ::= <element-ref> [',' <element-ref>]*
  <element-ref> ::= <element-prefix> '.' <element-name>
  <element-prefix> ::= 'control' | 'pane'
```

Data Binding

Keys that begin with `param` are serialized alphabetically in the data mappings file. These keys signify that the component in the value can be traced back to a specific property in a form parameter.

For example:

```
param_customer$firstName = control.fname
```

This signifies that the control **fname** is bound to the `firstName` attribute of the parameter `customer`. If the parameter type contains nested classes, the key consists of multiple property names, separated by `$`.

For example:

```
param_customer$address$zipCode = control.zip
```

Here, the key refers to the `zipCode` property of the `Address` object, which is contained by the `Customer` object.

In some situations, more than one component may reflect the value in the property of a form parameter. For instance, consider a master-detail pane where a property can show up as both a column in the master grid pane, and also as an editable field in the detail pane. In such cases, all components that can be traced back to the same data key are shown as a comma-separated list on the corresponding value.

For example:

```
param_customer$firstName=control.fname__master,control.fname
```

Bindings are also provided for panes:

```
param_customer=pane.customer
```

Here, the value of the `customer` parameter itself is bound to the **customer** pane.

Enabling the Generation of the Data Mappings File

The generation of the data mappings file is controlled by the project-level and workspace preferences.

If the generation is enabled and you remove or edit the data mappings file, the Form Designer regenerates it when the project or form is built. The corresponding `*.mappings.properties.json` file is deployed along with other form resources.

If you change the '**Generate a form mappings file**' preference, the '**Internal resource references are incorrect**' problem marker occurs on all the forms. You can fix it by using the **Quick Fix** context menu and by selecting '**Configure internal resource references**' on the Quick Fix popup.

Procedure

1. In the Project Explorer, right-click the project, and select **Properties**. Or click the **Project** menu, and select **Properties**.

   The Properties dialog for the *project name* opens.

2. In the left pane, click the **Form Designer** arrow to expand it, and select **Resources**.

3. Select **Enable project specific settings**.

4. Select **Generate a form mappings file**.

The check box is cleared by default.

5. Click **Apply**, and in the ensuing Rebuild? dialog, click **Yes**.

> You can also enable the generation of the Data Mappings file at the workspace level. Clicking **Window** > **Preferences** opens the Preferences dialog, in which, **Resources** is displayed in the expanded **Form Designer**. In the **Resources** pane, the **Generate a form mappings file** check box is available.

Bindings

TIBCO Business Studio Forms uses bindings to update properties in the runtime forms data model by connecting attribute values of parameters, controls, and panes. A binding always has two endpoints.

An absolute binding can connect the value of a control to the value of a parameter's data field, or to one of the child attributes or objects of that parameter.

Depending on the properties to be connected, bindings can be added from the **General Properties** tab of a control, pane, or a parameter.

An optionlist and radiogroup, a URL and URL Text of Hyperlink, and the URL of an Image control can also have bindings, which you can establish from the Properties tab of these controls. You can also use the **Mappings** tab to view, edit, and create bindings.

Click the **Add a Binding** button to set a binding for the given property or update that property using a rule that specifies a computation action.

Binding Between Controls

The **General Properties** tab for controls provides a mechanism for setting bindings between the value or property of one control and the value or property of another control or parameter.

When you define a binding for a control, its value is used to update the secondary properties of another control such as Label, Hint, and so on. The update is one way only, that is, the secondary properties cannot use bindings to update the value of the initially selected control.

Binding Between a Control and a Parameter

To connect a control with a parameter, you can use either the **General** tab of a control, or the parameter dialog for that parameter.

For information on working with bindings, see Setting Bindings.

Binding from the Mappings Tab

You can use the **Mappings** tab of the **Properties** view for selected element in the Form Designer canvas to set bindings. For more information, see Mappings Tab.

Direction of Bindings

A binding can have three directions.

The three directions are:

-
  Updated By :

  This signifies that the targeted value will be updated when the other value is updated.

  However, if the target value changes for any reason, the other value in the binding will not be affected.

-
  Update :

Updates to this value will cause the other value in the binding to be updated.

The control and parameter values can update other properties, but properties such as control visibility, enabled, required, label, and hint cannot update other values in a binding.

-
  Synchronizes With ⟨icon⟩ :
  With this type of binding, updates to either value will cause the other value to be updated to the same value. Each end of the binding must be either a control or parameter value.

Assigning Binding Both Ways

A two ways binding can be added for controls (only for values).

Procedure

1. Add a text control `textinput1`.

2. Add another text control `textinput2`.

3. Go to Properties tab of the control `textinput1` and click the binding icon for the **Value** field.

4. Search for `textinput2` control in the list and expand the items under it.

5. Click the **Value** field of the `textinput2` control.
   You will be able to assign a binding both ways.

Actions

Actions are invoked from rules in response to form events or programmatically from within a script. An action can be private to a single rule, or shared amongst multiple rules.

TIBCO Business Studio Forms uses three types of actions:

- **System actions**

  These actions, also called built-in actions, are pre-defined and are used for common tasks such as Submit, Close, Cancel, Reset, Validate, and Apply.

- **Script actions**

  Use JavaScript to create additional custom actions. Script actions run a specified script, with no other action attached to it.

- **Computation actions**

  These actions will update a specified value or property with the result of an expression written in Javascript. The destination of a computation action can be the value of a parameter or control, or a secondary property such as label or hint of a control, or a visible flag for a pane, and so on. After the script in the computation action is run, it produces a value that can be used by another action.

Actions can be flagged as "shared" allowing them to be used in multiple rules.

System actions can be used also by the users working in Business Analysis mode, while the scripted actions and computation action can be developed only by the users working in Solution Design mode. Once actions have been defined within a form by a developer, business analysts can re-use them for similar purposes in their projects.

To add an action, right click the **Shared Actions** system group in the **Outline** view.

To add and configure actions, see Setting Actions.

To associate actions with rules, see Setting Rules.

Actions Summary Table

The Actions summary table provides a useful overview of the shared actions. Clicking the **Shared Actions** node in the **Outline** view displays the Actions summary table in the Properties view, listing each shared action in the current project.

The Actions summary table displays the following columns:

- **Name**

  Name of the action. You can edit the name by clicking the ellipsis (...) button, which appears when you select the name. You can edit the name using the Enter the Name page.

- **Label**

  Label of the action.

- **Edit**

  Displays the text **Edit** as a hyperlink. When clicked, it navigates to the configuration property screen for that action.

- **Type**

  A non-editable field that shows either **ScriptAction** or **ComputationAction**.

- **Detail**

  A non-editable detail of the action specific to the action type.

  - ScriptAction display as much script as fits in the column, with "..." at the end if truncated.
  - ComputationActions display `[property] updated by expression: [script]`.

Rules

Rules provide a way to model the behavior or presentation logic of the form with minimal coding. This makes the logic easier to identify and maintain by both developers and business analysts.

Rules consist of events and actions. For example, the rule "Guardian required when Age < 21" is modeled as:

Event:

```
CustAge updated
```

Action:

```
GuardianName.Required = (CustAge < 21)
```

Whenever **Customer Age** changes, the **Guardian Name** field is marked as `required only if Customer Age is less than 21`.

Rules are associated with events and actions as follows:

- Events are used to trigger the rules, to define when the actions are performed. For any rules that are triggered by the same event, they will be executed in the order in which they are defined in the form model.

- Actions define what will be performed. They can be individually enabled or disabled in the rule. The actions within a rule will also execute in the order defined in the form model.

  > Business analysts can add rules, edit their general properties and descriptions, and add events. They cannot create new actions, but they can re-use the already defined shared actions.

You can add and edit rules in TIBCO Business Studio Forms as described in the following sections:

- Adding a Rule Using the Outline View: To associate rules with events and actions, select the appropriate **Events** or **Actions** tab.
- Adding a Rule Using the Rule Wizard: When using the **Rule** wizard, you can also remove the rule.
- To select actions and events to associate with a specific rule, see Setting Rules.

Rules Summary Table

The summary table for rules provides a useful overview of the rules.

Clicking the **Rules** node in the **Outline** view, displays the Rules summary table in the **Properties** view, listing each rule in the current project.

The Rules summary table displays the following columns:

Name

Name of the rule. To edit the name, click on the ellipsis **(...)** button, which appears when the name is selected. Edit the name using the Enter the Name page.

Label

Editable label of the rule.

Edit

Displays the text **Edit** as a hyperlink. When clicked, will navigate to the configuration property screen for that rule.

Enabled

Displays a check box. If selected, then the rule is enabled.

Cancel On Error

Displays a check box. If selected, then the form is cancelled on the occurrence of an error.

Events

Non-editable, drop-down list of events that trigger this rule; for example, `Form Open`, `Update of Control FirstName (firstName)`.

Actions

Non-editable, drop-down list of actions that are invoked by this rule. Each item will be in the form of `[Action Label] (Action Name)`.

The standard **cancel**, **close**, and **submit** actions destroy the form. You need to ensure that any user-defined actions for the **Cancel**, **Close**, and **Submit** button click event should precede their respective standard actions.

The Design Tab and Preview Tabs

The Form Designer in TIBCO Business Studio can have three tabs, the **Design** tab, the **GWT Preview** tab, and the **Mobile Preview** tab.

Each tab plays a different role:

- The **Design** tab is where you model your form and configure its properties.
- The **GWT Preview** tab shows how the form looks at runtime in a Google Web Toolkit (GWT) environment.
- The **Mobile Preview** tab shows the URL used to navigate and preview the mobile forms on a mobile device at design time.

  TIBCO Forms uses Google Web Toolkit (GWT) as the rendering technology for forms. The **GWT Preview** and **Mobile Preview** tabs are displayed or hidden based on the active runtime environment specified in the Presentation Channel preferences. See Configuring Presentation Channels for details.

The appearance of the form in the preview tabs is determined by settings that are configured on the property sheets of the form itself, and for the panes and controls within the form.

The **GWT Preview** tab act as working GWT application. You can specify data in the form, press the **Submit** button, and see the data that would be submitted to the server at runtime.

For example, if the user specifies a new customer name and clicks **Submit**, the **System Log** panel displays information about the specified text in GWT preview, if the INFO logging is enabled. To enable INFO logging, go to **Window - > Preferences - > Form Designer - > Preview** . GWT log samples are as follows:

GWT:

```
(-:-) 2011-08-18 11:15:49,242 [INFO ] **** Form Inout and Out Data ****
(-:-) 2011-08-18 11:15:49,242 [INFO ] { items:[{"$param":"text_field",
"mode":"INOUT", "type":"STRING", "$value":"John Smith"}]}
```

Thus the preview tab allows you not only to evaluate the appearance of your form with the current **Properties** view settings, but also to test its functionality.

Configuring Presentation Channels

The Presentation Channel preferences govern the runtime environment in which forms are built, previewed and deployed. These can be configured at project level or globally for all projects.

> If multiple form designers are working on the same project or projects, they should all have the same Presentation Channels configured in their respective workspaces.

For more information on Presentation Channels, see *TIBCO Business Studio Process Modeling Guide.*

Procedure

1. Select the project in the Project Explorer, and click **File** > **Properties** .

2. In the navigation pane on the left side of the Properties dialog, click **Presentation Channels**, and select the **Enable project specific settings** check box.

3. Double-click **Default Channel** (or other presentation channel you are using, if applicable) to edit the list of included channel types. You can have the following setting:

   - By default, Google Web Toolkit (GWT) environment is enabled. **Workspace Google Web Toolkit**, **Openspace Google Web Toolkit,** and **Openspace Email** check boxes are selected (**GWT Preview** tab is displayed)

   - To enable the Openspace Mobile environment, select the **Openspace Mobile** check box (**Mobile Preview** tab is displayed)

4. Click **Finish** and **OK** when you are done to close the dialogs. In Google Web Toolkit (GWT) environment, the changes take effect immediately just by refreshing or reactivating the preview tab.

   > To configure Presentation Channel globally, go to **Window** > **Preferences** > **Presentation Channels** . The **Default Channel (Default)** is displayed in the right side pane. Double-click **Default Channel** to edit the list of included channel types. The changes made at this level will apply to all projects that do not have the **Enabled project specific settings** check box enabled.

Port Settings for Preview

You can set the port used to serve up the preview of forms for both the internal preview tabs and the preview of mobile forms from external applications or devices. By default, this is set to 8888.

You can change the port if there is a conflict with another application using port 8888 on your machine.

To change the port, go to **Window** > **Preferences** > **Forms Designer** > **Preview** . If you change it to a value of 0, then an arbitrary, available port number will be used.

If you are using external devices such as mobile forms to test forms via the mobile index, it is recommended to keep this as a fixed port number so that you will be able to keep bookmarks to the mobile test index.

Copy Form Preview URL

By clicking the **Copy form preview URL** button in the main toolbar, you can copy the form preview URL to the system clipboard. You can then paste the URL in any browser to preview the form. This way you can see how the form is rendered in other browsers on a specific platform apart from the built-in browser used in Eclipse.

Logging

A system log pane for the preview tabs is provided to display trace and debug messages from the system as well as any logging messages from your JavaScript code.

The logging window displays the log output generated by the application, filtered according to the verbosity level set by the **Logging Level** list.

Locale

You can choose the locale from the drop-down list: English, Chinese, French, German, Spanish, and so on. Changing this setting only has an effect if locale-specific resource bundles are defined for the form.

For more info about localizing a form, refer to Localization of Forms.

The locale selected applies only to the form, not to the other components in the preview tab for instance the log window, **Locale** list, and so on.

Logging Level

For the GWT preview, the setting made in Preferences is the lowest level of logging available in preview. For example, if the logging level is set to INFO in the Preferences, you cannot change it to DEBUG in the preview pane.

The available log levels (GWT) are:

- FATAL
- ERROR
- WARN
- INFO (The default logging level)
- DEBUG
- TRACE

The verbosity (detail) of logging increases with the logging level in cumulative fashion. For example, the WARN level also shows all ERROR and FATAL messages; INFO also shows WARN messages; and so on.

You can choose the logging level in the preview pane using `context.form.log` or `context.form.logger`. The `logger` API is available in all the script contexts and it allows the user to log at all logging levels. See API for Scripting for details of `log` and `logger` APIs. The logging level specified applies only to that specific preview session. Messages logged by user scripts are shown in the DEBUG log level.

You can change the default logging level used in the preview tabs in the user preferences, under **Window** > **Preferences** > **Form Designer** > **Preview** .

At runtime, when GWT Forms are used, you can enable logging by using a URL parameter `log_level`. You need to set the value of the `log_level` parameter to any logging level. The specified log level is enabled in that case. For example, if you access Openspace as: `http://<server>:<port>/openspace?log_level=INFO`

You see INFO, WARN, ERROR and FATAL messages in the log viewer.

Reload

The **Reload** button in the GWT preview and in Mobile preview mode closes the current form and reloads it.

Performance Metrics

By default, the performance metrics option is disabled in GWT preview. The **Performance Metrics** button displays the form load timings.

You can view the performance metrics by pressing ALT+F12. The performance table is displayed with the timings for the operations listed in this table:

Performance Metrics Table

Column Name	Description
Overall Form Load Time	The time taken to load the form completely. It starts from the time a form is requested from the server and finishes at the time the form is loaded completely. This includes the Form Open scripts if any.
Model Initialization Time	The time taken to create and initialize the various form elements, such as parameters, panes, and controls. It does not include the time taken to load them with the initial data.
Resource Loading Time	The total time taken for various form resources to load. The resources include various external resources configured on the form and the generated BOM JavaScript files. The external resources include JavaScript, CSS, image, and property bundles referenced from the Resources tab in the Properties view of the form.
Library Resource Loading Time	The total time taken to load various library resources used by the form. The Resource Loading Time does not include this, but it is included in the Overall Form Load Time.
Form Rendering Time	The total time taken to render the form after the form model and various external resources are loaded. This does not include the time taken for creating various form elements, but includes the time taken for attaching the widgets, initializing the bindings, and loading the initial data to the form.
Datastore Initialization Time	The time taken for initializing the form elements from the initial data provided to the form.
Initial Deferred Rendering Time	The time taken to render the panes (and the components inside the panes) that are marked for deferred rendering but are visible on loading the form. This is not included in the Overall Form Load Time.
Post-Open Library Resource Loading Time	Tthe time taken for various library resources used by the visible deferred initialized panes to load during the form load but after the form open event. This is not included in the Overall Form Load Time and the Resource Loading Time.

You can use this information to analyze the load timings of various forms. This information is useful in UI automation and reporting.

In addition to the operations listed in the table, runtime also captures the time taken to destroy the form. All these timings are logged in the GWT log on destroying (that is, canceling, closing or submitting) the form.

It is displayed in the following format:

```
Perf Metrics: <form name and path>[<internal_form_id>], [instrument_id,
<total_time>, <start_time>, <end_time>]*
```

The Instrument IDs are:

- 1 - Overall Form Load Time
- 2 - Resource Loading Time
- 3 - Datastore Initialization Time
- 4 - Model Initialization Time
- 5 - Form Rendering Time
- 6 - Initial Deferred Rendering Time
- 7 - Library Resource Loading Time
- 8 - Post-Open Library Resource Loading Time
- 9 - Form Destroy Time

The start time and end time is relative to the start time of the form load.

Instrumentation Level

Instrumentation levels collect performance metrics of a form. There are four instrumentation levels - 0, 1, 2, and 3.

- The default is level 0, that is **None**, which does not collect any metrics
- Level 1, that is **Basic** collects the load timings of the form
- Level 2, that is **Call counts** collects the details of the number of times each action or validation script was executed in addition to the load timings
- Level 3, that is **Call times** collects the details of the duration of the execution of each action and validation script in addition to the number of executions and load timings

You can change the default instrumentation level from **Window** > **Preferences** > **Form Designer** > **Preview/Live Dev** by selecting an appropriate **Instrumentation Level**.

At runtime (Openspace, Workspace or a Custom Client Application), if you want to collect performance metrics, you can pass in a URL query parameter tibco_instr with a value 0, 1, 2, or 3. For example:

```
http://<server-host>:<port>/openspace/openspace.html?tibco_instr=1
```

With this URL query parameter, you can view the performance metrics anytime during the lifecycle of the form by pressing ALT+F12.

The earlier value of tibco_instr=true is now deprecated. Passing true is equivalent to passing 1.

View Datastore Data

The **View Datastore Data** button in GWT preview mode displays a preview of the current state of the form data that would be submitted to the server.

You can click this button at any point during form usage.

Visibility in the Preview Tab

All panes and controls are visible in the **Design** tab so that you can edit them, even if they are configured to be initially invisible at runtime.

For instance, the figure Invisible and Visible Form Parts is a form as it appears in the **Design** tab.

This form has panes with **Visible** property (on the **General** tab of the Properties View for each pane) cleared.

Invisible and Visible Form Parts

The shaded diagonal lines across two of the panes in this form indicate that the **Visible** property of those panes is initially cleared, or set to `false`.

In another section of the Capture Claim form, the visibility flag of the Witness Information pane is bound to the value of the **Witness Available** check box. When the check box is selected, the visibility of this pane is set to `true`, and the pane is shown. When the check box is cleared, the visibility of this pane is set to `false`, and the pane disappears. This behavior is fully functional in **GWT Preview**.

Outline View

While the Project Explorer provides an easy way to find, select, and open project resources, the **Outline** view provides a quick and convenient way to navigate within a particular model, such as a form.

If the **Outline** view is not visible, open it by selecting **Window** > **Show View** > **Outline** . (If Outline is not among the view choices, click **Window** > **Show View** > **Other** > **General** > **Outline** .)

The default area for the **Outline** view is the lower left corner of the Eclipse workbench but, as with other views, it can be moved to another area by dragging its title bar.

There are two modes for using the **Outline** view: as a hierarchical tree with expandable nodes, or as a thumbnail graphical image of the form. You can switch between the two modes by clicking the button for the desired mode in the upper right corner of the **Outline** view.

Forms Compact Mode

Forms compact mode has an additional mode that reduces the spacing between controls along with the spacing between labels and value fields within a control.

The `forms_compact_mode` common resource key controls the use of this mode.

Miscellaneous Resource Keys

Resource Key	Reference Value	Description
`data_preview_empty`	There is no data to display.	Used as a data preview message for empty data.

Resource Key	Reference Value	Description
`forms_compact_mode`	[1], [2], [3], [1,2], [1,2,3], [2,3], [1,3], or empty to disable the key	The key applies to all controls and panes on a form to make the form smaller in size. When the value contains 1, the width of the grid panes in the form is set to a maximum of 600 pixels. When the value contains 2, the labels align to the top even if the child labels are configured to be aligned to the left. However, if the pane has only controls in it, the labels are not aligned to the top. When the value contains 3, it reduces the spacing between controls along with the spacing between labels and value fields within a control. The default value for the run time is [3]. If you want to disable the compact mode, specify an empty value for it in the custom property resource bundle.
`align_toolbar_left`	true, false	Unused by default. Aligns the contents of the toolbar pane to the left when set to true. The toolbar buttons, such as **Submit**, **Close**, and **Cancel** are aligned to the right by default. For large forms that need horizontal scrolling, you can align the buttons to the left using this key.

See the *TIBCO Business Studio Forms User's Guide* for more information about resource keys.

Thumbnail Mode

The thumbnail mode shows the entire form scaled down to fit within the space designated to the **Outline** view.

When a form cannot be entirely rendered within the canvas, a blue-shaded rectangle appears in the **Outline** view representing the visible portion. You can drag this rectangle with the mouse to make a different portion of the form visible in the canvas. This is a good way to move quickly from one section to another of a large form.

Tree Mode

The hierarchical tree mode contains nodes for the form's elements. At the top level is a node for the form itself. The top-level nodes under the form are for the data interface to the form, shared actions, rules, and the root panes.

In the tree mode, clicking on an item in the **Outline** view causes the **Properties** view for that item to appear in the Properties tab , and causes that item to be selected in the canvas as well, if it is a visible object. This is a good way to move quickly to a particular **Properties** view. Items can be copied and pasted within the **Outline** view, as well as rearranged by using drag-and-drop.

Usage of the Outline View with Forms

When a form is open in the Form Designer, the **Outline** view's tree mode shows the elements that have been placed on the form, and provides a convenient way to select a pane or control and display its properties in the **Properties** view.

For instance, when a check box called **checkbox1** is clicked in the **Outline** view, the **checkbox1** control is selected on the canvas, and the **Properties** view displays the properties of that control.

There are situations where you may also find it easier to re-arrange the order of controls and panes in the form using the **Outline** view instead of the canvas, such as moving a control or pane to different locations in a large form where it is difficult to view the whole form in the canvas at once.

Although the order of Parameters, Shared Actions, and Rules in the form model does not have a bearing on the execution of the form, you have the option to arrange the order of these objects in the **Outline** view to aide in readability, or to group by functionality. By default, items are added to these nodes in the order they were originally added to the model.

Clicking on the Data node shows a summary table of all the parameters defined in the form. From this table, you can edit some of the properties, add new parameters, and navigate to the detailed **Properties** view of any of the parameters. Similar tables are displayed on clicking either `Shared Actions` or `Rules`.

Data

Data node, the first node beneath the form node in the **Outline** view, shows Data Fields and Parameters.

You can add new parameters or data fields from the context menu of **Data**.

Parameters

Clicking a parameter causes the **Properties** view for that parameter to appear in the **Properties** tab.

The **Context Menu** of a parameter allows you to delete, copy, or rename the parameter in the data model. Right-clicking on the Data node provides an option to add a new parameter.

Clicking a parameter in the tree mode of the **Outline** view is the only way to access the **Properties** view for the parameter.

For more details, see Configuration of Parameters.

Parameters Summary Table

The Parameters summary table provides an overview of the parameters. To see each parameter in the current project, select the **Data** node in the **Outline** view.

The Parameters summary table has the following fields:

- **Name**

  Name of the parameter. To edit the name, click anywhere within the Name cell and edit the contents.

- **Label**

  Editable label of the parameter.

- **Edit**

  Displays the text **Edit** as a hyperlink. When clicked, will navigate to the configuration property screen for that parameter.

- **Mode**

  Displays either IN, OUT, or INOUT. Specifies the direction of data flow for this parameter with respect to the Form.

- **Type**

  Displays the primitive type of the parameter. When selected, a drop-down list becomes available to choose among the predefined primitive types, or select External Reference to choose a BOM type from a user-defined business object model.

- **Length**

  Editable field for setting the length. It is active only if the selected type supports the length setting. Otherwise displays **NA**.

- **Decimal Places**

  Editable field for setting the decimal places attribute. It is active only if the selected type supports the decimal places setting. Otherwise displays **NA**.

- **Array**

  Check box that sets the array attribute of the parameter.

- **External Reference**

  Displays the external reference associated with the parameter. Clicking the external reference enables the picker button, which displays a list of available external references for that parameter.

Data Fields

Data fields hold private data for internal use by the form. You can consider them as a local or private scratchpad area.

Their values are not returned via parameters unless you specifically arrange for them to be returned. You can do this by using bindings, computations, or by passing them to `setValue()` API calls on parameters or components. You can use data fields in many ways, such as implementing OK or Cancel behavior for a modal dialog. For example, you may want to roll back the form changes on Cancel, or to commit them on OK by using data fields, events, and API methods.

The **Context Menu** of a data field allows you to delete, copy, or rename the data field in the data model. Right-clicking on the Data node provides an option to add a new data field.

Selecting a data field in the tree mode of the **Outline** view is the only way to access the **Properties** view for the data field.

Data Fields Summary Table

The data fields summary table provides an overview of the data fields. To see each data field in the current project, select the **Data** node in the **Outline** view, followed by the Data Fields tab in the Properties view.

The Data fields summary table has the following columns:

- **Name**

  Name of the data field. To edit the name, click anywhere within the Name cell and edit the contents.

- **Label**

  Editable label of the data field.

- **Edit**

  Displays the text **Edit** as a hyperlink. When clicked, will navigate to the configuration property screen for that data field.

- **Type**

  Displays the primitive type of the data field. When clicked, a drop-down list becomes available, and you can select one of the predefined primitive types, or select External Reference to choose a BOM type from a user-defined business object model.

- **Array**

  Check box that sets the array attribute of the data field.

- **External Reference**

  Displays the external reference associated with the data field. Clicking the external reference enables the picker button, which displays a list of available user-defined BOM types for that data field.

Shared Actions

Actions available to all rules are listed under the Shared Actions node. With the help of the **Context Menu** of the Actions node, you can add a new action to this group.

To read an overview, see Actions.

To learn how to add actions to a form, see Setting Actions.

Rules

Rules are listed under the Rules node. With the help of the **Context Menu** of the Rules node, you can add a new rule to the form. You can add a rule that is either enabled or disabled using this interface.

To read an overview, see Rules.

To learn how to add rules to a form, see Setting Rules.

Managing Form Elements From the Outline View

You can manage form elements in the **Outline** view, such as copy an element and and paste it on the canvas , or re-arrange the order of elements within the form.

You can rearrange form elements in the **Outline** view by dragging them and dropping them on the desired new place. The new arrangement will immediately be reflected on the canvas.

Procedure

1. Right-click the Form icon or any form element in the **Outline** view.

   The pop-up **Context Menu** appears.

2. Depending on the element selected different options are available, as explained in the table Manage Form Elements from the Outline View.

Manage Form Elements from the Outline View

Select	Definition
Cut (Ctrl+X)	Available for all elements except for fixed nodes (Form, Data, Shared Actions, Rules)
Copy (Ctrl+C)	Available for all elements except the fixed categories mentioned for 'Cut'. After you copy an element to the clipboard, you can paste it within this form or another form.
Paste (Ctrl+V)	Available when forms content is present on the clipboard
Delete (Delete)	Availble for all elements except for fixed nodes (Form, Data, Shared Actions, Rules)

Select	Definition
Rename (F2)	Available for all named elements.
Select All (Ctrl+A)	Selects all root panes. Select All will not select parameters, shared actions, or rules.
Show Properties View	Shows the **Properties** view, if not currently visible.

Use Business Labels in Outline View

The User Preference controls the display of labels throughout the Forms Designer.

This is specified using the option **Include type name in labels**, which improves accessibility by helping to distinguish the type of control or pane in various dialogs, instead of just relying on the icon. For more details on using this option, see Using the Option Include Type Name in Labels.

For more details about Labels, see Label.

Business Object Model

The business object model provides a way to define in business terms the Classes, Attributes, Primitive Types, Operations, Associations, and so on that describe a business or organization. In terms of forms design, the business object model is a powerful and convenient way of defining primitive and complex types.

A business object model is defined using the Business Object Model Editor. For complete information on using this editor to create business object models, see the *TIBCO Business Studio Business Object Modeler User's Guide*. Information on business object models in the present guide is limited to instructions for creating classes and other objects in the business object model to define complex data types, and using these data types in forms modeling.

The Objects in a Business Object Model

Objects are added to a business object model in the Business Object Model Editor much as panes and controls are added to forms, either by clicking the desired object in the palette and then clicking in the desired location on the canvas of the editor, or by dragging and dropping the object onto the canvas.

Objects that can be placed into a business object model include the Elements (Package, Class, Primitive Type, and Enumeration), Children (Attribute and Enum Literal), and Relationships (Generalization and Composition).

The objects in the palette are of several kinds, each distinguished by an icon and color, which appears (as an aid to the identifying the object) in various places throughout the Business Studio interface, including in the title bars of the objects on the canvas. The objects most important for creating complex types to be used in forms modeling are described in this section.

Elements

Class

A container for a complex data object. Classes contain children, such as attributes and enum literals. A class from the BOM can later be specified as the type for a data field in the Forms Editor.

Primitive Type

An object of one of the BOM Primitive Types (Integer, Boolean, Date, Time, Integer, and so on), or of the type of a previously-defined primitive type object.

In the latter case, the previously-defined primitive type might be, for instance, a zip code object that was defined as an integer with a pattern (specified in the **Advanced** tab of the object's **Properties** view) as a regular expression) that limits its value to 5 single-digit integers. The Pattern value restricts valid entries to five integers. This restriction is enforced at runtime.

⚠️ A pattern that has been specified as a restriction for a data type in the BOM does not appear in the Forms modeling environment. For instance, if a **ZIP Code** primitive type is defined in the BOM as requiring a value of five single-digit integers, and that primitive type is included in an **Address** class in the BOM which, in turn, is used as a data type for a form parameter, the default generated form will not display the restriction in the **Validations** tab of the zip code text control's **Properties** view. Nonetheless, the restriction will be enforced at runtime, and cannot be modified or overwritten by different restrictions defined in the Forms Editor on the text control's **Properties** view.

Enumeration

A data type that can contain a list of values. Selecting this type enables you to specify a set of enumerated values. For example, an enumeration called Color might have the values Red, Blue, and Green.

An enumeration from the BOM can be included as an attribute for a class in the BOM or be specified later as the type for a data field in the Forms Editor. On the default generated form, this type will be rendered by default as an optionlist. (The control type could later be changed in the form control's **Properties** view to a radiogroup, or other control type.)

Children

Attribute

Attributes are data members that make up a class. By default, new attributes are created with the primitive BOM type `text`. A different data type can be chosen in the attribute's **Properties** view, either another primitive type, or an existing class or enumeration. Each attribute type ends up corresponding to a different control type in a generated form.

The attributes in a class can be re-ordered in the **Attributes** tab of the class's **Properties** view using the up and down arrows. Their order in the BOM determines the order in which they appear in the default form.

Enum Literal

These are the values within an enumeration. For example, an enumeration called Color might have the enum literals with the names Red, Blue, and Green.

The enum literals in an enumeration can be re-ordered in the **Enum Literals** tab of the enumeration's **Properties** view using the up and down arrows. Their order in the BOM determines the order in which they appear in the default form.

Relationships

Generalization

This is a relationship of inheritance: a class that is related to an existing class by generalization will inherit the qualities of the existing class, and hence will contain members of the same type as the existing class.

Composition

This relationship indicates that the child class is wholly contained within the parent class.

Multiplicity of Relationships

Relationships between BOM classes have a *multiplicity*, for instance, one-to-one (1..1), zero-to-many (0..*), or one-to-many (1..*). You can also have a finite lower or upper multiplicity bound like one-to-finite upper bound (1..m), finite lower bound-to-finite upper bound (n..m), or exactly finite bound (n).

On a generated form, a particular pane type is rendered for a child class based on the multiplicity value.

If a Student class, for instance, has a child class called Course, with a 0..* relationship (meaning that one student can have many courses), the Course class will be rendered as a grid pane. The attributes of the Course class (for instance, course number, course name, time, room number, and so on.) will appear as columns in the grid pane. Each course for a given student will be represented by a row in the grid pane.

Implicit Validations

The multiplicity constraints defined in the BOM are reflected in the implicit validations. The validation messages conform to the following:

Validation Messages for BOM Level Multiplicity Constraints

Multiplicity Constraint	Validation Message
One-to-many (1..*)	Must contain at least one value.
One-to-finite upper bound(1..m)	Must contain between one and {m} values.
Finite lower bound-to-finite upper bound (n..m)	Must contain between {n} and {m} values.
Zero-to-finite upper bound (0..m)	Must contain between zero and {m} values.
Exactly one (1)	Must contain exactly one value.
Exactly equal to the finite bound (n)	Must contain exactly {n} values.

These apply for both primitive attributes and complex children.

The implicit validations for multiplicity constraints are configured to execute on form submit.

Master-Detail Panes

If a child class has a relationship to the parent class that allows multiple instances of the child class, and the child class itself contains a child class with multiple attributes, the two child classes will be rendered on the default form in a master-detail pane.

The first child, the master pane, will be rendered in the form as a grid pane, and the second child, the detail pane, will appear as a vertical pane which can be used for editing all attributes of both child classes.

If you want the detail pane to be generated as a record pane, go to **Preferences** > **Form Designer** > **Generator**, and select the check box **Generate master-detail configuration with record pane for details**.

By default, the check box is cleared, and the detail pane is generated as a vertical pane. This information applies to the default forms and newly generated forms. The forms that are already generated, remain unaffected.

In this case, the grid pane will be read-only, but a row can be selected for editing in the vertical pane (detail) by clicking that row in the grid pane (master).

As an example, a `Student` class might be the parent of a child class called `Course`. Each student could have zero-to-many courses. The course class, in turn, might have a child class called `Course Details`. The BOM diagram is shown in the figure Business Object Model Editor Showing Child Classes.

Business Object Model Editor Showing Child Classes
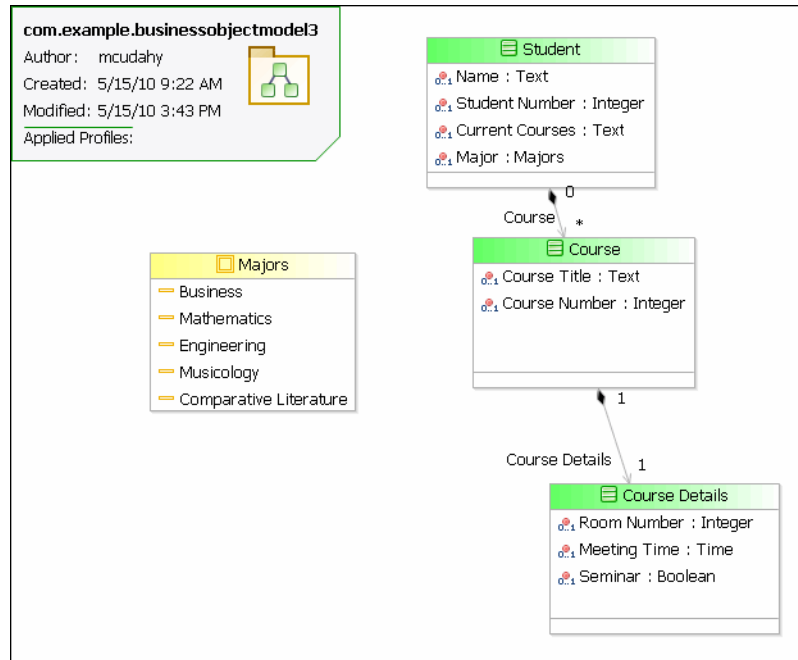
The business object model shown in the figure Business Object Model Editor Showing Child Classes is rendered in a form with a master-detail pane for the `Course` and `Course Details` classes.

Selecting a row in the grid pane (that is, the master pane) allows that row to be edited in the vertical or record pane (that is, the detail pane). An alternate way of selecting rows for editing is to enable navigation for the record pane. Navigation is turned off by default, but is enabled by selecting the **Show Navigator** check box in the **Properties** tab of the record pane's **Properties** view. The navigator then appears for the record pane.

With navigation enabled, you can delete the grid pane from the form if you consider it unnecessary to provide users with two methods for selecting records to edit. However, you cannot do this for the vertical detail pane, as it is single-valued, and thus does not provide a navigator. You can manually refactor the detail pane from vertical to record, and then bind it to the correct data.

Live Development of Forms

With the BPM Live Development capability, a BPM client such as Openspace runs using the form resources in the local TIBCO Business Studio workspace rather than those previously deployed to the server, without the need for redeployment.

This way, you can iteratively make incremental changes to the form and instantly see the results in the context of the real-life deployed application, using live process instance data. When using the BPM Live Development mode, you can change the user interface of a form, but not its data interface. You cannot change other deployable models such as process packages, organization models, and business object models. If you do update other assets such as processes, organization models, or business object models, you need to redeploy the project to the run time.

The BPM Live Development (BPM Live Dev) perspective contains the TIBCO ActiveMatrix Openspace view hosting a browser session connected to Openspace and a Properties view for setting the Live Dev session preferences. When this perspective is active, the Project Explorer view is automatically filtered to show only form-related resources, and the Form Designer actively prevents changes to data interfaces of the form. Once the form is loaded in the BPM Live Dev perspective, you can load the latest changes to the form in the workspace by using the **Refresh** button.

⊘ The BPM Live Dev perspective is not supported on mobile Openspace and accessible Openspace.

Viewing Forms in BPM Live Development

Use BPM Live Development when you want to make quick changes to elements of your project, then test the results immediately without having to redeploy an entire project. This is especially relevant to Forms development, where small changes require a rapid turnaround for retesting.

Modifications to assets like process, BOM, and organization model are not supported in BPM Live Development. If you change those types of assets, the project needs to be redeployed. If you change the Forms data interface, the project must be redeployed or the form will not open.

If you are running Openspace on an HTTPS server, you may need to disable protection in order to load the form. This is due to the mixing of the HTTPS content from the server and the HTTP content from TIBCO Business Studio.

Procedure

1. Deploy the entire BPM application in the default **BPM Modeling** (or **Modeling**) perspective (if not already deployed).

2. Switch to the **BPM Live Dev** perspective (select **Open Perspective** > **BPM Live Dev** in the top right of the pane).

3. Log in to Openspace (in the embedded view, or open an external browser).

   Any BPM server can be used, not just the Local Development Server. (For more information about the Local Development Server, see *TIBCO Business Studio BPM Implementation*.)

   You can edit the information in **Openspace View Connection** for **Openspace/Client Base URL** and then refresh the view for the development server.

   The Openspace view provides a **Copy openspace url** button that copies the openspace URL, which can be pasted into an external browser to log in to Openspace in **Live Dev** mode.

4. Start a process and proceed until the required work item arrives (or start a business service and progress to the appropriate form).

5. Access a form on opening a work item/business service. The form from your design-time workspace is used instead of the deployed one.

6. Iteratively:

   - Test the form.

   - Edit the form in the local workspace and click **Save**.

   - Either use the **Refresh** button provided on the form, which reloads the form with the latest changes without the need to re-open it or restart the pageflow for instance, or close the work item or business service in Openspace. A **Cancel** button is also provided on the form in cases where the Form loading fails. The **Cancel** button cancels the form if the form fails to load.

   - Reopen the work item / business service in Openspace.

   The process flow works as normal and you can complete/edit data in the same way as normal in Openspace.

Setting Preferences for BPM Live Dev

You can set preferences for the BPM Live Dev perspective at the workspace level.
The following preferences are available for BPM Live Dev:

- **Local host (Live Dev)**: By default, the host name of the machine hosting the local TIBCO Business Studio instance is selected. It lists all the DNS host names configured on the local machine, which

can be passed to the remote server. You can also refresh the list of DNS host names using the **Refresh** button next to the list.

This host address is passed to Openspace and must be resolvable and reachable from the computer that hosts the Openspace server. By default, Openspace connects to Studio through the local loop-back interface to the host `localhost/127.0.0.1`. If Openspace is hosted remotely, virtually or both, you must set the local host to an address that is resolvable and reachable from the Openspace server. If the network configuration of the local machine is multi-homed, the list contains all the host names and IP addresses from those network interfaces through which the configured Openspace host is reachable. To support a remote Openspace server, the firewall of the local computer must accept incoming HTTP connections on the configured preview port.

- **Render accessible user interface** - It is not selected by default.

  🚫 | The BPM Live Dev perspective is not supported on accessible Openspace.

- **Render enhanced user interface** - By default, forms in the old format are not rendered with the enhanced UI, and forms in the new format are rendered in the enhanced UI. You can force the enhanced user interface by selecting **Always**, or disable it entirely by selecting **Disable**.

- **Channel Type** - The default channel is used. You can explicitly specify **Desktop** or **Mobile** to force the use of a particular channel type, however, it is discouraged. You can use it to force mobile forms to render on a desktop browser and vice versa, but it may not work well with all the browsers.

- **Logging Level** - By default, no logging is done for BPM Live Dev. You can specify any of the following logging levels:

  - None

  - Fatal

  - Error

  - Warn

  - Info

  - Debug

  - Trace

- **Instrumentation Level**: By default, performance metrics is not collected. However, you can specify either of the following levels:

  - None

  - Basic

  - Call counts

  - Call times

For more information on the instrumentation levels, see Performance Metrics.

Setting Preferences from the Properties View

You can set the preferences for BPM Live Dev from the **Properties** view of Openspace.

Procedure

1. Open the Openspace view in the BPM Live Dev perspective.

2. In the **General** tab of the **Properties** view, specify preferences under **Forms Live Development**.

3. Click the **Refresh** button under **Openspace View Connection** so that the changes you made take effect.

Setting Preferences from the Preferences Dialog

You can set the preferences for the BPM Live Dev perspective from **Window** > **Preferences** .

Procedure

1. Open the Preferences dialog by clicking **Window** > **Preferences** .

2. Expand **Form Designer**, and select **Preview / Live Dev**.
   Available preferences for preview and BPM Live Dev are displayed in the right panel.

3. Specify the preferences, click **Apply**, and then click **OK.**

Cross-Resource References

The Business Studio workspace acts as a container for resources such as projects, folders, and files, each of which corresponds to a directory or file in the operating system's underlying file system.

Workspace files can contain models (such as forms or business object models), which are comprised of model elements (such as panes and controls or classes and properties).

A form can refer to model elements in other resources in the Business Studio workspace, for example:

• A user task or its parameters

• A business object model class or its properties

• An embedded form or its parameters

These references are often many-to-many, with one form referencing many external model elements and resources, each of which could potentially be referenced from multiple forms, business object models, processes and so on. These external references are known as cross-resource references.

Since the referenced model elements reside in independently modifiable files such references are susceptible to breakage if proper working procedures are not observed. When Business Studio detects breakages, it creates **unresolved reference** problem markers on the referencing forms.

For more information, see Breakage Mechanisms and Quick Fixes.

Breakage Mechanisms

There are several ways in which a cross-resource references can be broken.

Some examples are:

• The referenced model element could be deleted

• The referenced model element could be renamed

• The element's containing resource, folder or project could be deleted, renamed or moved elsewhere

When such changes are made using Business Studio, it attempts to prevent reference breakage by cascading such updates through all references. For example:

• In the case of rename and move of an element or a containing resource, the references are all automatically updated to point to the new element name or workspace location.

• In the case of deletion of a cross-referenced workspace resource, Business Studio presents a confirmation dialog offering the choice of clearing or retaining the references or cancelling the delete command. Clearing the references means that the connections between referenced and referencing elements are permanently severed and can only be restored manually.

> In most cases such changes might prevent the referencing forms from working as intended and can cause other problem markers to appear if it places the forms into an invalid state.

We now discuss some breakage scenarios in detail.

Deletion of an Embedded Form

When an embedded form is deleted, you are offered a choice of either clearing the reference or retaining it.

- Clearing the references to a deleted embedded form leaves the embedded form panes in an invalid state because they no longer point to a form to embed.

- Conversely, retaining the references means that the referencing forms are left pointing at a resource or model element that no longer exists in the workspace, which will cause **unresolved reference** problem markers to appear.

The confirmation dialog presented by Business Studio when any form-referenced resource is deleted can be suppressed by selecting the **Do not ask this question again** check box on the Clear Forms References dialog.

In this case, in future by default all the references are cleared.

If necessary, you can still use the **Preview** button, and deselect any **Clear forms references to deleted elements** changes.

Whether it is appropriate to clear or retain the references depends on your intentions.

- If you are deleting the resource because it is no longer required you should probably clear the references. In this case you would have to edit the forms to restore functionality.

- If you are deleting the resource with the intention of reinstating it later, it is probably appropriate to retain the references. However, if you do this the form will be left in an unusable state and all manner of errors and problems would ensue if you tried to work with it.

Considerations for Making Changes to Business Studio Resources

Cross-resource references can also get broken by editing, renaming, moving or deleting resources without Business Studio's knowledge, for example by changing the files directly in the underlying file system.

References can also get broken by making changes in one workspace and copying only a subset of the affected resources into another workspace.

⚠ These practices are strongly discouraged but unfortunately it may not always be obvious that a given action runs the risk of breaking a reference.

The basic principle is that related projects and the resources they contain are densely interconnected and should therefore be treated as an indivisible whole, managed exclusively from within Business Studio.

Problems with Business Studio Project Export/Import Wizard

Some development teams try to use the Eclipse File System or Business Studio Project Export/Import wizards to share projects or individual files and folders.

⚠ This practice is not recommended, as project-level exchange is at once too coarse-grained for convenient team development (where different developers make incremental changes to individual resources) and/or too fine-grained to maintain the integrity of cross-resource references and dependencies.

For example - if you move or rename a BOM file that is referenced from another project, this will update all forms references including those in referencing projects. If you then export just the project containing the changed BOM and import it to another workspace, the referencing forms in the target workspace will acquire **unresolved reference** problem markers because they will still be pointing to the old BOM file name or location.

If you have to use project export/import, you are recommended always to transfer a consistent set of projects, where all dependencies can be resolved from within the export/import location. Similarly, when importing projects, be sure to import all their dependencies as well.

Remember that you will be unable to import a project that already exists in the workspace and that the existing project may be inconsistent with the remaining visible incoming projects.

Advantages of Using Eclipse Team Providers

There is really only one satisfactory way for a development team to share resources, which is to place all projects under version control managed by an Eclipse team provider.

Business Studio bundles the Subclipse team provider for Subversion for this purpose. Many other version control systems have Eclipse team providers, which may or may not work well with Business Studio projects. Business Studio assumes optimistic version control concurrency semantics, so it does not support team providers which create read-only working copies or require an explicit working copy lock prior to editing (such as Perforce).

Even so, team members must take care not to do things which affect resources being modified by other team members – if this happens a *merge conflict* will result. The most reliable way to resolve a merge conflict is the 'optimistic locking' approach of rejecting one change set in its entirety then reapplying the rejected changes to the accepted change set. Otherwise, you will be faced with a tricky, error-prone textual merge of complex XML model files.

Quick Fixes

If a reference does get broken, Business Studio provides several quick fixes.

- **Reload the working copy** quick fix removes stale **unresolved reference** problem markers.
- **Clear the reference** quick fix simply clears the offending reference.
- **Repair the reference** quick fix helps you to locate a suitable replacement model element.

Reload the working copy - Quick Fix

This quick fix is used to remove the **unresolved reference** markers that can sometimes linger after the missing resource has been reinstated; this can sometimes happen during project import.

Clear the reference - Quick Fix

This quick fix can be applied to multiple **unresolved reference** problem markers simultaneously. It simply clears the offending references, which often places the referencing form model into an invalid state that is then reported by other problem markers. Such problems must then be fixed individually from within Form Designer.

Repair the reference - Quick Fix

This quick fix can only be applied to one **unresolved reference** problem marker at a time. It presents a dialog that lists all the possible model elements that could be used as a replacement for the missing referenced model element.

The Repair Reference dialog has a set of filters that allow you to broaden or narrow the scope used to identify potential matches. When the dialog first comes up, all filters are active and no candidate items are visible. You can selectively disable filters to broaden the match scope until the list of candidates includes the desired replacement. The dialog remembers the filter settings. You can also type part of the target element name in the search box at the top the list will be filtered to show just the elements which match the search string. The filters are:

Project name

When this filter is active the list shows only matching items from the same project as that containing the originally referenced element. If no project of that name exists in the workspace you will have to deselect this filter to see anything at all.

File name

When this filter is active the list shows only items which reside in a file of the same unqualified name as that containing the originally referenced element. If no file of that name exists in the workspace you will have to deselect this filter to see anything at all.

Element type

When this filter is active the list shows only items which have the same type as the originally referenced element. For example, if the originally referenced element was a BOM class, the list will only show BOM classes. It is recommended to leave this filter enabled.

Element qualifier

When this filter is active the list shows only items which have the same qualifier name as the originally referenced element. For example, if the originally referenced element was a BOM type or property, the qualifier is the containing BOM package, so the list will only show BOM types or properties from a BOM package of the same qualified name as the original.

Element name

When this filter is active the list shows only items which have the same unqualified element name as the originally referenced element. For example, if the originally referenced element was a BOM type or property, the element name is the unqualified BOM type or property name (not the label).

Selecting the desired replacement and pressing the **OK** button closes the dialog and updates the form to point to the selected element, and the **unresolved reference** marker goes away. If the chosen item is in an unreferenced project the wizard requests permission to add a project reference.

Alternatively, pressing the **Clear** button closes the dialog and clears the **unresolved reference** – see the description for the **Clear the reference** quick fix.

Delete the model element - Quick Fix

This quick fix cascade-deletes the model element that holds the unresolved reference. That is, depending on the actual element type, it deletes either the element itself or the nearest containing model element whose removal would restore the form model to consistency.

For example, consider the end-point of a binding that references an object which no longer exists. This end-point holds an unresolved cross-reference. If the quick fix were to delete only the offending end-point, the binding would remain broken, as one of its two mandatory end-points would be missing. So, the quick fix cascade-deletes the entire binding rather than just the offending end-point.

In many cases, cascade is not necessary and the quick fix removes only the element bearing the unresolved reference.

Mobile Forms

TIBCO Forms is designed to provide rendering suitable to the device used to access it. Mobile forms functionality of TIBCO Forms ensures optimized rendering on mobile devices.

TIBCO Forms is supported on Apple iOS devices. The supported platform is iOS 7.

You can design mobile forms by configuring the controls specifically for mobile usage. The **Mobile Preview** tab is provided to view mobile forms at design time: you can type the URL specified in the **Mobile Preview** tab in the mobile device's web browser to access the form.

Due to space limitations on a mobile screen, mobile forms are displayed one pane at a time. If the form has nested panes, they are shown as links. You can use the **Back** button on the form to navigate back to the containing panes in the form.

Most of the functionality available on the desktop version of forms is supported on the mobile version. However, there are some features which are not supported currently and few controls behave differently on mobile devices. The limitations are as follows:

- The settings on the **Layout** tab and the **Font** tab in the **Properties** view of controls are not supported.

- The settings on the **Child Labels** tab and the **Child Layout** tab in the **Properties** view of the pane is not supported.

- The **Label Visibility** flag on the **General** tab in the **Properties** view of controls and panes is not supported.

- The **Hint** field on the **General** tab in the **Properties** view of controls is not supported.

- The **Maximum Length** and **Display Length** fields on the **Properties** tab in the **Properties** view for text controls are not supported.

- Custom controls are not supported.

- The **Pass-through** control is not supported.

- **Multi-select Grid** panes are not supported.

- Modal dialog panes are not supported.

- Static and deferred rendering are not supported. The panes are rendered as ordinary panes.

- The BPM Live Dev perspective is not supported on mobile forms.

Modified Functionality

Some of the panes and controls function differently when they are rendered on a mobile device.

See Rendering of Mobile Forms for more details.

- Horizontal panes are displayed as vertical panes

- Message panes are ignored. Messages are displayed under each control instead of the message panes. If the control is inside a nested pane, the pane links in the form indicates errors if there are errors inside its controls.

- Grid Panes are edited only via master-detail pane pattern.

- Certain data entry controls such as Date, Time, DateTime, Duration, and Optionlist behave differently.

Enabling Mobile Forms

You have to enable the **Openspace Mobile** channel type to activate mobile forms.

You can enable mobile forms globally within the workspace or for specific projects in your workspace. You can enable the **Openspace Mobile** channel locally within a project by going to **Context Menu** > **Properties** > **Presentation Channels** > **Enable project specific settings** .

Procedure

1. Go to **Window** > **Preferences** > **Presentation Channels** .

2. The **Default Channel (Default)** is displayed in the right pane.

3. Select the **Default Channel (Default)**, and click the **Edit Item** ✏️ button.

   The Presentation Channel dialog is displayed.

4. Select the **Openspace Mobile** check box from the list.

5. Click **Finish.**

6. Click **Project** > **Clean** to clean the project.
   This will activate mobile forms.

Result

Once mobile forms are activated, the **Mobile Preview** Tab is displayed in the editor.

Mobile Forms Preview

The **Mobile Preview** tab provides the URL used to navigate and preview the forms on mobile at design time.

The URL is in the following format: `http://<host>:<port>/forms/mobile`

where:

- `<host>` is the name or IP address of the machine on which TIBCO Business Studio is running.

- `<port>` is the forms preview port. By default the port is 8888. To change the port, go to **Window** > **Preferences** > **Forms Designer** > **Preview** .

Typing this URL in an iPhone, iPod touch, or the iPhone emulator available from Apple takes you to a page that provides a list of the projects in the workspace. Drilling down in a project, displays a list of the forms available in that project.

The iPhone emulator runs only on Mac OS. There are no viable emulators available in Windows. You can use the desktop version of Safari to view forms on a Windows machine. However, certain controls (Date, Time, Date Time and single select Optionlist) do not function in the desktop version of Safari.

Mobile Specific Configuration of Controls and Panes

When you are designing a form for mobile devices, you need to configure specific pane and control properties.

You can configure the following properties:

Mobile Specific Configuration of Pane and Control Properties

Property	Configuration and Behavior
Short Label	Used to specify a short label which is displayed instead of the label for the mobile rendering of the form. All controls and panes support a Short Label. To set the Short Label, go to the **Mobile** tab in the **Properties** view of the component and specify the Short Label. The Short Label can be updated via the API, bindings, or computation actions.

Property	Configuration and Behavior
Toolbar Pane	Used to mark one pane as the toolbar pane in a form which is targeted for mobile devices. Mobile Forms adds a toolbar at the top of the page. You have to set a pane in your form as a toolbar pane so that it can be rendered in the toolbar area. A toolbar pane must be the root pane and only one toolbar in your form must be targeted for mobile devices. A toolbar renders the controls horizontally, so it is recommended to use only 3 button controls in toolbars. Toolbars typically provides a set of actions to the user, so you should only have button controls in them. A navigation pane in the form is automatically set as toolbar pane. To set the toolbar pane, go to the **Mobile** tab in the **Properties** view of the pane and select the **Toolbar** check box. This toolbar pane is rendered at the top of the screen. To set the maximum number of buttons controls go to **Preferences** > **Form Designer** > **GWT Forms** > **Maximum mobile toolbar buttons** .
Start Year	Used to specify the first year that should be displayed in the date picker in mobile forms. To set the Start Year, go to the **Properties** tab in the **Properties** view of the date and datetime controls. The default value is `-20`. The value specified in the Start Year determines the earliest year to display. The value specified is either an absolute value or relative to the current year when the form is viewed depending on the Start Year Relative field settings.
Start Year Relative	Used to specify whether the value of Start Year is interpreted as being relative to the current year or as an absolute year. To set Start Year Relative, go to the **Properties** tab in the **Properties** view of date and datetime controls. The default is `true`. If this is set to `true`, then the value of Start Year is interpreted as being relative to the current year. The value specified is added to the current year to determine the earliest year to display.
End Year	Used to specify the last year to be displayed in the date picker in mobile forms. To set the End Year, go to the **Properties** tab in the **Properties** view of the date and datetime controls. The default value is `20`. The value specified in the End Year determines the latest year to display. The value specified is either an absolute value or relative to the current year when the form is viewed depending on the End Year Relative field settings.
End Year Relative	Used to specify whether the value of End Year is interpreted as being relative to the current year or as an absolute year. To set End Year Relative, go to the **Properties** tab in the **Properties** view of date and datetime controls. The default is `true`. If this is set to `true`, then the value of End Year is interpreted as being relative to the current year. The value specified will be added to the current year in determining the latest year to display.

Property	Configuration and Behavior
Minute Increment	Used to specify the increment to use when displaying the choice for minutes in a time or datetime control. To set Minute Increment, go to the **Properties** tab in the **Properties** view of time and datetime controls. The default value is 15 and the maximum value is 60. For example, a value of 10 will display choices of 0, 10, 20, 30, 40, 50. A value of 60 will only display 0 as a choice.

Rendering of Mobile Forms

A few controls behave in a different way when they are used in mobile forms and rendered on a mobile device.

The differences are as follows:

Date Control

The pane that contains the date control displays the formatted date. On selecting the date, a date spinner is shown that allows you to select day, month, and year. The range of years is bounded and is configured in the **Properties** tab in the **Properties** view of the control.

Date Spinner
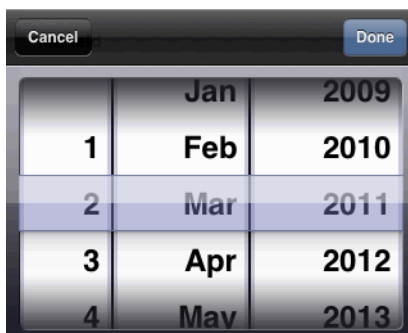
Time Control

The pane that contains the time control displays the formatted time. Selecting the time displays a time spinner that allows you to select hour and minute. The selector uses a 12 hour spinner with AM/PM.

Time Spinner
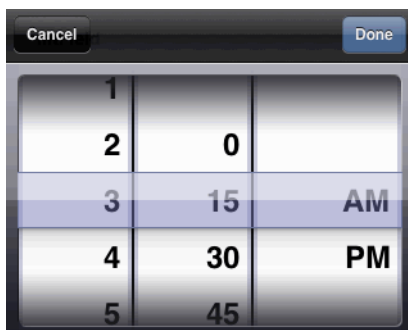
Datetime Control

The pane that contains the datetime control displays the formatted date and time. On selecting datetime, you go to the next screen where the date and time are displayed as two separate links. You can click on the date and time links to set them individually. Clicking the **Back** button will take you back to the previous screen.

Duration Control

The pane that contains the duration control displays a read-only summary of the information. Clicking on the control displays a detail screen where values can be specified for each of the fields.

Duration Control

Image Control

The pane containing the image control has a link for the image. Clicking on the link takes you to the next screen that displays the full image.

Optionlist Control (Single Value)

The pane that contains an Optionlist control shows the label of the selected option, clicking on which shows a choice spinner from which you can select a choice.

Choice Spinner

Radiogroup Control

Radiogroup controls are converted to optionlist controls in the mobile version of the form.

Textarea Control

The pane containing the textarea control displays the label. You can select the control to see the text area appear in a full screen. Selecting the **Back** button returns to the parent pane.

Horizontal Panes

Horizontal panes are converted to vertical panes in the mobile version of a form.

Record Panes

Record panes are used at runtime to handle both grid panes and record panes. The record pane supports all navigation functionality such as go to first, previous, nth, next, and last record. You can navigate to a specific record using the spinner control. The plus and minus icons on the navigation bar are used to add and delete records.

Record Panes Display
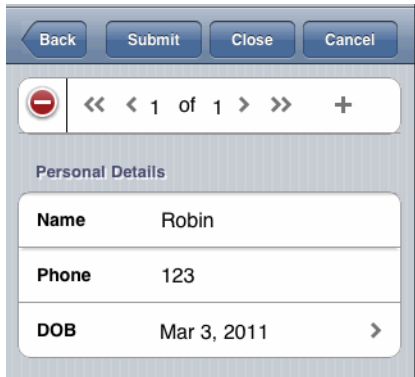
The navigation bar in a record pane displays information on which records in the record pane have validation errors.

Tabbed Panes

Tabbed panes are represented as vertical panes with each of the tabs being a nested pane. It will therefore be displayed in the UI as a list of links to the individual tabs.

Problem Markers

Problem markers are a standard Eclipse feature that track issues associated with workspace resources.

They appear in the **Problems** view, which can be filtered in various ways, as well as on elements in the **Outline** view and in the Form Designer. A marker includes a summary of the problem and identifies the affected file and the internal location. It also has a severity level (error, warning, or informational). The marker icons indicate the severity level:

	Error
	Warning
	Informational

Double-clicking a form validation marker will open or activate the Form Designer and select the offending form element (generally a pane or control). You can then use the **Properties** view or canvas to fix the problem manually.

Quick Fixes

Some of the problems detected by the Validation Builder can be corrected automatically by applying a Quick Fix. If a Quick Fix is available, the corresponding action on the problem marker's **Context Menu** is enabled.

With the Quick Fix dialog, you can select the fix to apply (there may be more than one), and also select other instances of the same problem in order to fix them all at once. You can do this only for non-interactive quick fixes.

The Quick Fix dialog inherits the filter settings from the **Problems** view. The dialog displays other instances of a given problem that could be fixed by the selected Quick Fix, but only those which are visible in the **Problems** view. For example, to fix all instances of a given problem within the enclosing project or the entire workspace, you may need to select **Configure Contents** action from the **Problems** view menu and change the **Configuration** or **Scope and Severity** filters.

Tasks

Using TIBCO Business Studio Forms, you can create user-friendly forms by changing the layouts, by configuring panes to have various components, and by setting rules for validating input data.

Creation of a New Form

The forms, created using TIBCO Business Studio Forms, are associated with a process task or a pageflow task, or are embeddable in other forms.

There are several ways to create a new form in TIBCO Business Studio.

- Go to the context menu of the `Forms` special folder, or any folder under the Forms special folder in the Project Explorer and click **New** > **Form** .

- On the **File** menu, click **New** > **Other** > **TIBCO Forms** > **Form** .

- Go to the context menu of a user task in a business process, and click **Form** > **Open** .

- On the **General** tab of a user task's **Properties** view, select the **Form...** radio button.

Of these approaches, the first two are equivalent. Both of these approaches trigger the opening of the New Form dialog.

You need to specify the **Form type** on the New Form dialog. The type of form that is selected here determines the components that are initially part of the form model. The form types details are as follows:

- **Process task**: This creates a form that is the same as one created from a User Task in a process definition. It will contain a root pane, a toolbar with **Cancel**, **Close**, and **Submit** buttons, and a messages pane for displaying error messages.

- **Pageflow task**: This creates a form that is the same as one created from a User Task in a Pageflow Process. The only difference to a Process task form is that the toolbar contains only **Cancel** and **Submit** buttons. The **Close** operation is not supported in pageflows since there is no way to re-open a step in a pageflow once it has been closed.

- **Embeddable**: This creates a form that is suitable for embedding within another form. This only contains a single root pane. This is because the parent form typically contains the toolbar and messages pane, so these components are not needed in an embeddable form.

The other two approaches are equivalent. They will generate a form that has parameters and a user interface component corresponding to each of the parameters in the user task interface. For more information on creating a new form for a user task, see *BPM Implementation Guide*, Chapter 4, Using Forms for User Tasks.

Drag and Drop Gesture to Customize a Form

You can customize a default form or create a free standing form by using the drag and drop (DND) gestures supported by the Form Designer. With these gestures, you can quickly add new user interface items onto the form canvas.

- From the **Project Explorer** view, you can use the DND gestures for:

  - Business Object Model (BOM) class

  - User task parameters

  - Process datum (Parameter, Data Field)

  - Form files

DND Items from the Project Explorer View
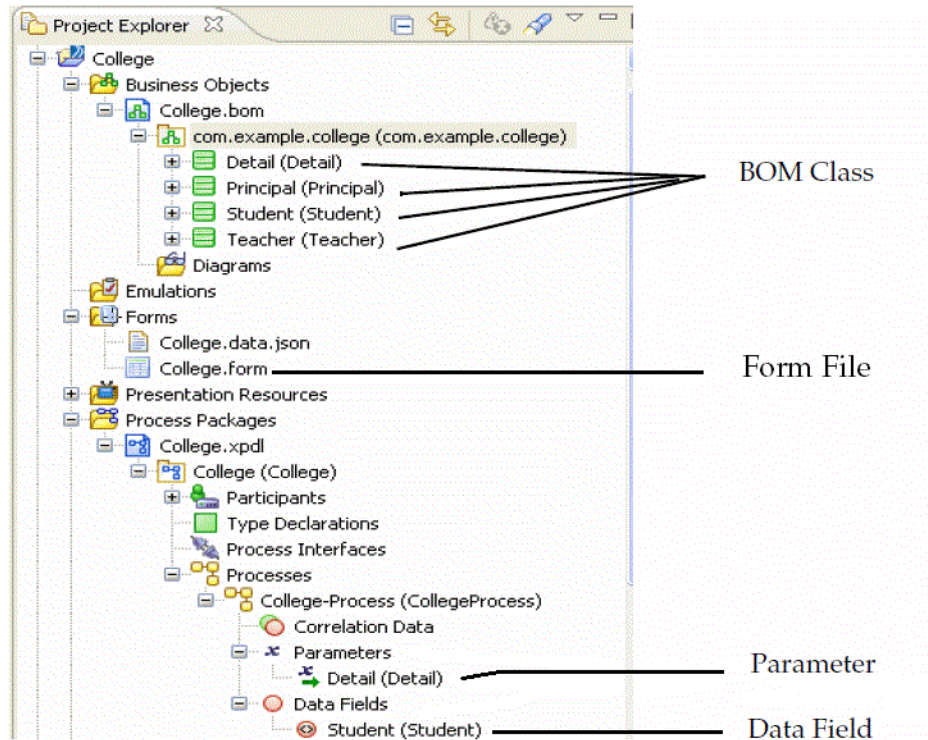
- From the Form Designer **Outline** view, you can use the DND gestures for:

  – BOM property

  – Form datum (Parameter, Data Field)

> The BOM property can only be dropped onto a pane that is associated with a BOM class that actually owns or inherits the dropped property.
>
> Using the DND gesture for BOM property is very helpful in restoring any missing user interface items in the form.

DND Items from the Form Designer Outline View
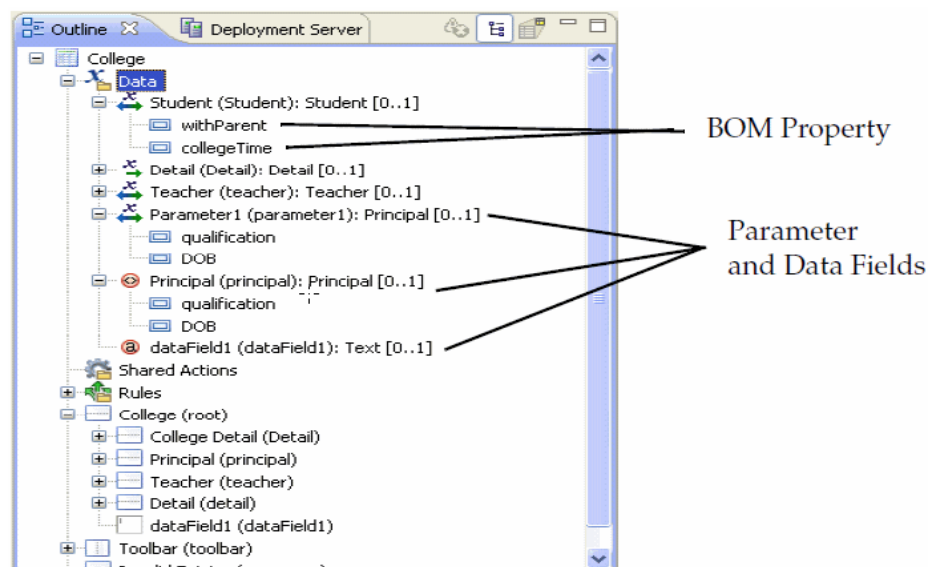
The drop gesture results in the creation of any or all of the following, as appropriate:

- A matching form parameter is created, if no matching parameter exists. This applies only to the Project Explorer drags.

- A suitable user interface component (control or pane with child components) is created, if none already exists.

- Bindings from the new or implied form datum and its children to the generated user interface component and its children are created.

For Project Explorer DND:

- the drop handler matches an existing parameter if one with the same generator source or of the same name already exists

- If not, it creates new parameters of type corresponding to the dropped objects

Matching is performed on the basis of whether a parameter exists that was originally generated from the same model as is being dropped, or failing that matching on type.

DND UI creation is essentially a form synchronization operation. The form synchronizer attempts to create any missing components within a hierarchical UI structure that matches that of the underlying data. If you heavily modify a form and move components around to a point where the synchronizer cannot identify the UI component (or ancestors thereof) corresponding to a dropped UML property, it re-creates the UI structure matching the data. You can then move the newly created components of interest to the appropriate location in the form, safe in the knowledge that any bindings will be automatically refactored. You can also safely delete any extraneous components.

The new form model elements are created by the standard form generator and thus follow the same generation rules. If attached to an existing generated form structure, they also become candidates for subsequent sync validation and synchronization.

When dragging from **Project Explorer** view it is important to drag the most appropriate model element. For example, if you are working on a form for a user task, drag the user task parameter, or (if these are not explicitly modelled) drag the process parameter or data field. If you are working on an embeddable form, drag the BOM class. Dragging a BOM class onto a form intended for use with a user task may produce a deceptively correct-looking User Interface. However, this interface is with the BOM class rather than the user task parameter, process parameter, or process data field. This may lead to ambiguity and unexpected results in subsequent synchronization operations.

Setting Bindings

For most controls, many properties on the **Properties** view can be initialized by an inbound parameter or expression.

The properties that may be initialized in this way are identified by the presence of an **Add a Binding** button to the right of the field where the property's value is set.

As explained in Bindings, the options for adding a binding are:

- From the **General Properties** tab for a control

- From the parameter dialog for a specific parameter

- From the **Mappings** tab of the **Properties** view for the selected element

You can also set bindings from the **Properties** tab of the properties sheet for some controls, such as hyperlink.

For an overview of bindings and their use in TIBCO Business Studio Forms, see Bindings.

Adding a Binding from the General Properties Tab for a Control

The **General** tab of the Properties view for a control may contain the binding icons indicating that a parameter or expression can be bound to any of the following properties: **Label**, **Hint**, **Value**, **Visible**, **Enabled**, and **Required**, which each can have only one binding or computation action.

The value property can have multiple bindings and/or computation actions. For details about these properties, see General Tab.

Procedure

1. Click the **Add a binding** button next to a property.

   The Select Type dialog appears.

2. Select the radio button **Create a binding for this property**.

3. Click **Next**.

4. In the Edit Binding dialog, configure the binding as explained in the table Edit Binding from the General Properties Tab for a Control.

   Edit Binding from the General Properties Tab for a Control

Select	Definition
(Down arrow above the Select an Items text box)	Click the Down arrow on the right (above the Select an item... window) to select from these options: • **Show controls and panes** If this is not selected, then only parameters will be shown in the Matching and selected items pane. • **Show unbound items only** If this is selected, then any properties that already have bindings will not be shown. You can select either one, both, or none by clicking on the corresponding check mark.
Select an item	This text box allows you to type in a filter expression that will restrict the items shown in the Matching and selected items pane. Names, labels, and property names are matched by the filter. You can use the * and ? wildcard characters to represent any string or any character respectively.
Matching and selected items	In the Matching and selected items list, select a property to which you want to bind the initially selected property. This selection appears right under the Matching and selected items list as a complete path to the selected property: *../pane/control/property* For example, select the parameter (`CustAge`), which will update the Guardian Name if the customer age is less than 21.

Select	Definition
Define the binding type for the selected property	In the section *property* of *control*, the three binding directions are displayed: • Updates *property* of *control* • Is updated by *property* of *control* • Synchronizes with *property* of *control* The binding types that are available for use are enabled, while the ones that are not available appear as disabled (grayed out).
Select Binding Endpoint window	If the selected binding type for the specified property is not allowed, an error will appear in the Select Binding Endpoint window.
Finish	If the selected property can be bound the way it was selected, the **Finish** button in the bottom of the diagram is enabled.

5. Once the binding configuration is finished, all new binding icons appear next to the property.

Adding a Binding from the Parameter Dialog

The **General** tab of the Properties view for a parameter contains a binding icon indicating that a parameter can be bound to a control.

Procedure

1. Select the property in the Outline View, such as **Name (CustName)**.

   The **General Properties** tab for the value **Name** is displayed.

2. Click the **Add a binding** button next to a property, such as for the Label property of the Name control.

   The Select Type dialog appears.

3. Select the **Create a binding for this property** radio button, and click **Next**.

4. In the Edit Binding dialog, configure the binding as explained in the table Edit Binding from the General Properties Tab for a Control.

5. Once the binding is configured, it appears next to the property.

Adding a Binding from the Mappings Tab

The **Mappings** tab of the **Properties** view for a selected element provides a comprehensive view of all the bindings and computation actions. You can view, edit, and create bindings from the **Mappings** tab.

Refer to Mappings Tab for further details.

Removing a Binding

You may want to remove a binding before deleting or moving the element elsewhere.

Procedure

• Click the **Remove** button in the Edit Binding dialog.
  The binding will be removed along with the icon in the general tab.

Setting Actions

You can set either a script action, or a computation action.

For an overview of actions and their use in TIBCO Business Studio Forms, see Actions.

Adding a Script Action Using the Outline View

You can add a script action from the **Outline** view.

Procedure

1. In the context-menu of **Shared Actions**, select **New Script Action**.

2. Type or select data as explained in the table Specify Details to Define a New Script Action.

Specify Details to Define a New Script Action

Field	Description
Name	Type the name for the new action.
	The name is only visible with the Solutions Design capability. It must be unique among all actions in the form and comprised only of alphanumeric characters and the underscore "_". The name may be referenced from the JavaScript of other actions when using the `invokeAction` method.
Label	Type a descriptive label for the new action.
	The Label is used in other parts of the Form Designer to identify the action. It is not used at runtime.
Script	In the Script window, type the script for the new action.
	See Scripting for a discussion of the variables available in this script.

3. Click **Finish**.

The new script action button is displayed in the **Outline** view indicating a shared action.

Adding a Computation Action Using the Outline View

You can add a computation action from the **Outline** view.

Procedure

1. In the context-menu of **Shared Actions**, select **New Computation Action**.

2. In the Enter the Action Details dialog, type or select data as explained in the table Specify Details to Define a New Computation Action.

Specify Details to Define a New Computation Action

Field	Description
Name	Type the name for the new action.
	The name is only visible when the Solutions Design mode is active. The name must be unique among all actions in form and must be comprised only of alphanumeric characters and the underscore "_". The name may be referenced from the JavaScript of other actions when using the `invokeAction` method.
Label	Type a descriptive label for the new action.
	The Label is used in other parts of the Form Designer to identify the action. It is not used at runtime.
Destination	Click the Browse **(...)** button to select a property to update with the results of the script evaluation.
	Once you select the value, it will appear in the Destination window, such as `Value of Claim Amount (ClaimAmount)`.
Expression	Type the script that will be evaluated in order to update the property selected in the **Destination** field.
	This is a JavaScript expression. The expression may contain multiple lines, but the last line in the script must be an expression that will be used to update the destination.
	Note:
	Do not use a return, since you are not writing a function.

3. Click **Finish**.

    The new script action button is displayed in the **Outline** view indicating a shared action.

Editing an Action

You can modify script and computation actions that are shared by selecting them in the **Outline** view and specifying the changes in the **General Properties** tab for that action.

Setting Rules

By setting rules on event triggers, you can make the form more responsive.

For an overview of rules and their use in TIBCO Business Studio Forms, see Rules.

Adding a Rule Using the Outline View

You can add a rule from the **Outline** view.

Procedure

1. In the context-menu of **Rules**, select **New Rule**.

2. In the Rule Details page of the New Rule dialog, specify data as explained in the table Specify the Details for Rules.

Specify the Details for Rules

Field	Description
Name	Type the name for the new rule.
	The name is only visible when the Solutions Design mode is active. The name must be unique among all actions in form and must be comprised only of alphanumeric characters and the underscore "_". The name may be referenced from the JavaScript of other actions when using the invokeAction method.
Label	Type a descriptive label for the new rule.
	The Label is used in other parts of the Form Designer to identify the rule. It is not used at runtime.
Enabled	Enable (default) or disable the new rule by selecting or clearing the check box.
	If disabled, the actions defined in the rule will not be executed, even if the one of the rule events is triggered. This option is provided primarily as an aid in debugging a form.

3. Click **Next** to define the rule.

    In the Rule: Pick Events page, use the **Add** button to add events or the **Delete** button to remove events associated with the rule.

4. Click the **Add** button.

    The Select Event page , which is used to select the events that trigger a rule, opens with a dialog Select Item.

5. Click the event you want to associate with the rule, such as update property. You may select multiple events by holding down the control key as you select the events.

6. Click **OK** to confirm the selection.

    You can add multiple events to the rule. You can also delete any of the previously associated events from the list.

7. To define an event, click **Next** in the Rule: Pick Events page.

    The Define Actions page opens.

8. Click **Add**.

9. In the Pick an existing action or choose the create a new one dialog, there are two choices:

    - Picking an Existing Action
    - Creating a New Action

Picking an Existing Action

You can pick an existing action to define an event.

Procedure

1. Click the Browse button (...) next to **Pick an existing action**.

    This will allow you to choose one of the system actions, or to select one of the custom shared actions defined in the form.

2. In the Select Item dialog, select an action from the list of Matching and selected items, and click **OK**.

A new row appears in the table with the details of the action.

3. Click **Finish**.

   The Define actions dialog appears.

4. In the Define actions dialog, you can further configure the new action by selecting (or clearing) the check boxes to enable (or disable) the action, or to designate the action to be shared.

5. Use up or down arrows to move the selected actions and rearrange them in the window.

   The actions will execute in the defined order when the rule is triggered by one of its events.

6. Click **Finish.**

Creating a New Action

You can create a new action to define an event.

Procedure

1. Click the **Create a New Action** radio button.

   Two additional radio buttons become available: **Script Action** and **Computation Action**.

2. Select the type of action you want to create.

3. Click **Next**.

4. If you selected Script Action, specify the data as in the table Specify the Action Details for the Script Action:

   Specify the Action Details for the Script Action

Field	Description
Name	Type the name for the new rule.
	The name is only visible when the Solutions Design mode is active. The name must be unique among all actions in form and must be comprised only of alphanumeric characters and the underscore "_". The name may be referenced from the JavaScript of other actions when using the `invokeAction` method.
Label	Type a descriptive label for the new rule.
	The Label is used in other parts of the Form Designer to identify the rule. It is not used at runtime.
Script	Type the script to run.

   If you selected Computation Action, specify the data as in the table Specify the Action Details for the Computation Action:

Specify the Action Details for the Computation Action

Field	Description
Name	Type the name for the new rule.
	The name is only visible when the Solutions Design mode is active. The name must be unique among all actions in form and must be comprised only of alphanumeric characters and the underscore "_". The name may be referenced from the JavaScript of other actions when using the `invokeAction` method.
Label	Type a descriptive label for the new rule.
	The Label is used in other parts of the Form Designer to identify the rule. It is not used at runtime.
Destination	Click the Browse icon (...) to select a value of the property to update based on the script evaluation.
	Once you select the value, it will appear in the Destination window, such as `Value of Claim Amount (ClaimAmount)`.
Expression	Type the script that will be evaluated in order to update the property selected in the Destination field.

5. Select the **Shared** check box on the actions dialog to create shared actions from your new custom actions. This makes actions visible under shared actions in the **Outline** view and available for use in other rules.

Adding a Rule Using the Rule Wizard

In addition to adding new rules through the **Outline** view, you can also create a computation rule (a rule with a computation action) directly from property you want to create a computation rule for.

Procedure

1. Click the **Add a binding or rule** button next to a property, such as for the `Value` property of the **Name** control.

2. In the Select Type dialog, select the **Update this property using a Computation Action** radio button.

3. Click **Next**.

   In the Provide Expression dialog, provide the script. In this case, there is no option to select the destination, since the destination is implicit on where you are adding the computation action.

4. Click **Next**.

   In the Events Configuration dialog, use the **Add** button to add events or the **Delete** button to remove events associated with the rule.

5. Click the **Add** button.

   The Event Picker, which is used to select the events that trigger a rule, opens with a dialog Select Item.

6. Click the event you want to associate with the rule.

7. Click **OK** to confirm the selection.

   You can add multiple events to associate with the rule. You can also delete any of the previously selected events from the list.

8. Click **Finish** when you are done configuring the rule.

    A rule icon appears next to the property, and allows easy editing of the compute action. Additionally, the rule appears in the **Outline** view and can be edited as a regular rule.

Enhanced User Interface

TIBCO Forms uses an enhanced user interface from the 3.1.0 release. The default styling of forms is now enhanced using the widely accepted Bootstrap styles.

In addition to the current set of custom CSS classes, the forms are rendered with a specific set of pre-defined classes applied by the run time.

In the case of projects deployed from a previous version of TIBCO Business Studio (up to 3.8.0) to its compatible run time (up to 3.0.0), the forms are not rendered in the enhanced user interface. If you redeploy the project after upgrading to the latest release, its forms are rendered in the enhanced user interface.

You can enforce the enhanced user interface using the URL parameter, **`tibco_enhanced_ui`**.

The values permissible for the parameter are:

*   `default`: All the newly deployed forms are rendered in the enhanced user interface, while the forms deployed in previous releases (up to and including TIBCO Business Studio 3.8.0) are rendered as before.

*   `enable`: All the forms, including the older ones, are rendered in the enhanced user interface.

*   `disable`: All the forms are rendered in the old interface. If you are opening the forms in a new window every time, you must pass the value as part of the parent window URL.

If the URL parameter is missing, the run time considers a meta declaration for `gwt:property` in the host HTML file with the name `tibco_enhanced_ui` with the same set of values as mentioned above. For example,

```
<meta name="gwt:property" content="tibco_enhanced_ui=enable"/>
```

If both the URL parameter and the meta declaration are missing, the run time considers the value defined in the common bundle of the respective forms. For example,

```
tibco_enhanced_ui=enable
```

You can enhance the UI that uses Bootstrap, or integrate another CSS framework by making changes to the default configuration properties in the common resource bundle as follows:

*   By overriding the specific entries for each component type in the common resource bundle
*   By adding style declarations in the relevant CSS file

The keys for static panes are different than the keys for the controls in a static pane. Also, there are different keys for list controls.

For each pane and control on a form, two keys are used:

*   A generic key for a pane with the name `pane_class_defs`
*   A generic key for a control with the name `control_class_defs`

You can use a generic key `control_custom_class_defs` to write class definitions for custom controls, which are similar to the class definitions for built-in controls.

The format for the second key is `control_<control_type>_class_defs`, where the `<control_type>` is the last part of the control type in the model. For example, for the control type "com.tibco.forms.controls.textinput", the key is `control_textinput_class_defs`. For the form itself, the key is `form_class_defs`.

For information on control types, see Control Type.

Definitions for each control or pane are in a JSON format, that is a JSON array containing JSON objects with the following properties:

- **selector**: Mandatory. This property specifies the path of an element inside the control or pane from its top level `div` unless specified explicitly using the `target` property. The path must be an absolute path in the DOM. You need not provide the top level node. If you do, start the expression with `//`. In the expression, you can use the HTML attributes in the DOM using the syntax `[attribute=value]`.

- **classes**: Mandatory. This is a space-separated list of CSS classes to be applied on the element matching the selector expression.

- **when**: Optional. The value can be valid or invalid, specifying when to add the respective classes. If not specified, the classes are always applied on the element.

- **where**: Optional. It is applicable only to controls. The value can be `in-grid` or `not-in-grid`. If it is `in-grid`, the classes are applied only when the control is in a grid. If it is `not-in-grid`, the classes are applied only when the control is not in a grid. If not specified, the classes are always applied irrespective of where the control is.

- **target**: Optional. Its value can be `widget`, `label`, or `list-edit-widget`. When not specified, the path in the selector starts from the outer `div` of the component. If the path has to start from the widget, the value must be set to `widget`. Similarly it must be set to `label` if the path is taken from the label. The value must be set to `list-edit-widget` if the path is taken from the edit widget in a list control. With the `target` property, you can define styles specifically for when the control is in a grid or not in a grid.

> Both, controls and panes have a top-level `div` that contains two `divs` - one represents the label and the other contains the actual widget, such as an input box for a Text control and its corresponding hint.

To know more, see Keys for Enhanced User Interface and Customizing Property Resource Bundles.

Enhanced User Interface on Custom Clients

The runtime bundles the Bootstrap, but does not load it by default. You can include the Bootstrap by using the exported API, tibco.forms.Util.loadCSS().

The Bootstrap is bundled in the default theme with path `bootstrap/bootstrap.min.css`. If the containing application needs to load Bootstrap, you can call the above API from the onTIBCOFormRunnerLoad() callback. For example,

```
tibco.forms.Util.loadCSS("bootstrap/bootstrap.min.css");
```

For more information, see Utility Methods.

CSS Best Practice

CSS best practices are enforced while creating or editing a form, with the help of a work-space level preference **Enforce CSS best practice**. It hides the Font, Layout, Child Layout, and Child Label properties, and disables the resizing gestures on the canvas.

You can disable this preference from **Window** > **Preferences** > **Form Designer**.

Defining explicit overrides to fonts, colors, borders, and sizing of form elements results in generating *<form>*.css or explicit HTML attributes, thus violating the HTML best practices. If the existing forms contain such overrides, a warning marker appears in the Problems view, saying "`explicit styling violates CSS best practice`". Applying the quick fix restores the enforcement of CSS best practice.

Cascading Style Sheets

It is possible to specify additional CSS classes that are applied to form components at the form, pane, and control level.

This approach provides more flexibility and opportunities for reuse of style information than manually setting properties at the Form model level. You can control some layout and font properties using the form model Property tabs.

Setting CSS Classes

The **General** property sheet for the form, panes, and controls includes an input box to specify the CSS class for the given component. The value can be either a single value, or a space-separated list of CSS classes. When the component is rendered in the web page, the CSS classes specified here are added to the HTML along with other built-in CSS classes. The value of the CSS class for a form, pane, or control can also be updated using bindings, computation actions, or set via the API.

Using an external CSS resource

The **Resources** property sheet for a form allows one or more external CSS files to be referenced from the form. When added as an external reference, the CSS is loaded prior to the loading of the form. To load an external CSS file in a form:

- Place the CSS resource within the Presentation Resources special folder.
- Select the root of the form by either clicking in the background of the canvas or selecting the root node in the Outline view.
- Select the "External Resources" Properties view.
- Click the **(plus)** button to add a reference to the CSS.

See Scripting for lists of the built-in static and dynamic CSS classes.

> Internet Explorer 8 and 9 have a limitation on the number of CSS files that can be loaded by a webpage. Due to this limitation, some of the custom forms with additional CSS may not render as expected.

Best Practices

Follow these best practices for better stylistic consistency and code reuse.

- Use `.TibcoForms` in class selectors.

  The root node of each form specifies the `TibcoForms` class. You can then write CSS selectors that are specific to Forms and that do not conflict with other elements on the page. For example, suppose you have a CSS class **highlight** that you apply to a pane. The corresponding CSS rule may be written as follows:

  ```
  .TibcoForms .highlight {background-color: yellow;}
  ```

  This ensures that the **highlight** class gets applied only to elements within a form.

- Share CSS between forms.

  You can share the same CSS between multiple forms to avoid duplication. Just add a reference to the shared CSS from one or more forms.

- Avoid introducing new custom CSS classes if the desired styling can be achieved through selectors using the built-in CSS classes.

  For more information, see CSS Classes.

Examples

A vertical pane might make use of a set of classes such as:

```
pane pane-vertical
    pane-label
    pane-content
        component control-textinput required
            label
            tf-container
                widget-text
                hint
        component control-date
            label
```

```
          tf-container
              widget-date
              hint
```

The following selectors may be useful:

.pane-vertical .hint

  Applies to hints within vertical panes

.control-date .label

  Applies to labels of Date controls

.pane-vertical .required

  Applies to required controls within a vertical pane

.pane-vertical .pane-horizontal .label

  Applies to the labels of controls and panes in horizontal panes nested within vertical panes.

Data Validation in a Form

TIBCO Forms supports runtime validation of data as the user fills up the forms. You can configure validations for the fields defined on the form. You can configure the validations to occur either when the user changes a field value, or when the user submits the form.

Validations help users to specify correct data, thereby enhancing the overall experience. On the server side, the submitted data are validated against the restrictions specified in the business object models used within the form.

You can write validation scripts for each control as well as each pane on a form. Validation scripts usually run when users update data or submit the form. The scripts need to be written to explicitly return a Boolean or an Array.

* If the returned value for all validation scripts on the form is `true`, the form data are valid.

* If the returned value for one or more validation scripts is `false`, the validation error messages are displayed on the form in a special pane called a Messages pane. Users can click the error message to navigate to the first instance of the error in the form.

* If the validation expression evaluates to an array of strings, it indicates a failed validation. In this case, the Messages pane substitutes each indexed parameter marker in the validation message template with the corresponding array element.

The Messages pane displays the validation messages. You can specify a validation message either using a key reference from the **External Resources** of a `*.properties` file or as a **Custom Message**.

By default, the Messages pane opens at the bottom of the form when a validation fails. By manually adding a Messages pane to the form, you can configure the font and layout properties of the pane, and place it anywhere other than the default position.

Validation Messages and Usability

Good validation messages help users complete the forms faster and without any error in the specified data. Users get to see error messages for various types of controls and panes.

If you configure validation messages for each control, the user gets the validation message for the control after specifying data and moving on to the next control. If you configure the validation to occur on submitting the form, the validation message appears after the user clicks **Submit** to submit the form.

Clicking the message associated with an individual control sets the focus on that control. Record and Grid panes automatically navigate to the correct page in order to show the invalid control. If you configure validation for the entire pane, the focus of the screen shifts to the beginning of the pane in case of a failed validation.

For information about invoking validations programmatically, see `validate` in the API reference.

Validation Script

The final expression in the validation script must evaluate to `true` (if the data are valid), `false` (if the data are invalid), or an array of strings (if the data are invalid and the validation message contains substitution variables).

In the Edit Validation Script dialog, you can edit the script that determines whether the data submitted are valid, or you can modify the error message that appears when users submit invalid data.

You can use the notation `this` in your script to refer to the control or pane during a given validation invocation. A validation script, for instance, might contain a statement such as:

```
this.getValue() == "New York";
```

You can also use the context object provided while executing the validation to retrieve the value of the given control or pane:

```
context.value == "New York";
```

You can refer to any control by using the "`control.`" notation, or to a pane using "`pane.`" notation. To refer to the value of a control, use the latter notation in conjunction with the `Control.getValue()` method:

```
control.city_name.getValue() == "New York";
```

Validation scripts must have no side effects. Do not set the value of controls nor make any modifications to the form model from within a validation script.

Implementing Validations

You can add, edit, or remove validation scripts only when using the **Solution Design** capability. If the **Solution Design** capability is disabled, the **Validations** tab does not appear on the Properties view of a control.

You can enable or disable a validation at the time of defining it, or after defining it. If disabled, the validation definition remains in the form model, but is not invoked at runtime. This may be useful during troubleshooting of a form.

- When defining a validation, you can enable it or disable it by using the **Enabled** check box on the Define Validation dialog.

- You can enable or disable a defined validation by using the **Enabled** check box in the **Validations** tab of the **Properties** view.

Adding a Validation

TIBCO Forms does not validate controls and panes that are invisible, disabled, have any empty value, or that are contained within a pane that is invisible or disabled. Only collection panes are validated even if they are empty.

Procedure

1. With the form open in the Form Designer, click the control or pane where you wish to add new validations.

   The Properties view shows the properties for that control or pane. You can view the validation script for any control or pane by clicking the control or pane, and clicking the **Validations** tab in the Properties view of the control or the pane. See **Reference** for a detailed description of each property available on the **Validations** tab.

2. Click the **Add** button to add a new validation.

   The Define Validation dialog opens.

The Define Validation Dialog

3. Specify a unique name for the script in the **Name** field.

4. Select the **Execute When** option from:

   a) **On Form Submit**: Sets the validation script to run when the user submits the form. When more than one control is involved, such as when you want to ensure that at least one of the two or more fields are filled in, you can select **On Form Submit**.

   b) **On Value Change**: Sets the validation script to run when the user specifies a value in the field, and then exits that field. The validations of the syntax of specified values are best performed **On Value Change**.

5. If you are defining a validation on a pane or control that supports multiple values (for example, grid panes, list controls, and multi-select optionlists), select **Validate As List** to control how the validation is run.

   > If you select **Validate As List**, then the validation runs just once for the entire list of items, and `context.value` contains an Array (for primitive values) or a list (for multi-valued pane validations). If you do not select **Validate As List**, then the validation runs once for each item in the multi-valued control or pane, with `context.value` set to a new item each time the validation is invoked.

6. Specify the validation script in the **Script** text area.

7. Select the type of **Message** from:

   - **External Reference**: Picks the validation message from an external `*.properties` resource. You can define validation messages at the form level in an external resource file with `validation_` as a prefix in the key, and share the file across forms or projects. Also, the default implicit validations can reference messages in the common resource bundle. External reference

validation messages can use substitution variables to include runtime data values in an externalized static text string.

- **Custom**: Allows you to specify custom text message or a message that contains substitution variables, for example: "Sorry, you cannot have more than {0} {1}". You can dynamically determine the validation message at the runtime using substitution variables.

8. If you select the **Message** type as **External Reference,** click the picker button to open the Resource Picker dialog. Select a validation message from all the available `validation_*` resource keys, and click **OK**.

> The Resource Picker dialog displays a filtered list of only `validation_*` resource keys.

The Resource Picker Dialog
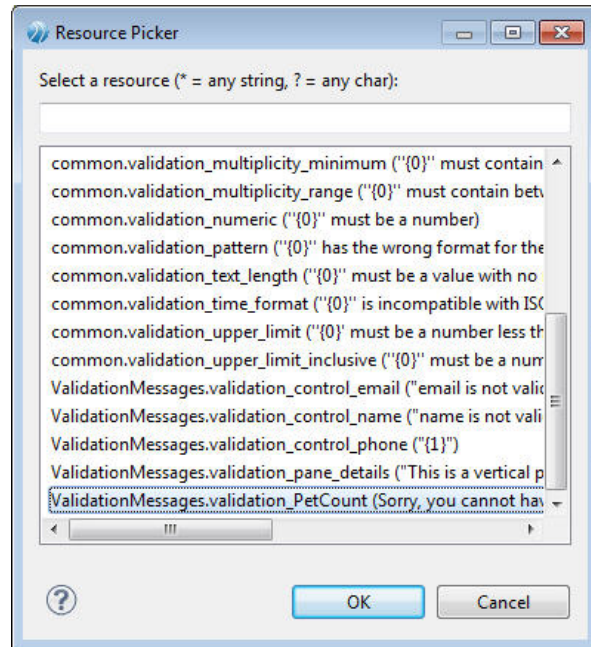
See Example 3 Validation Message Referenced from External Resource for details.

9. If you select the **Message** type as **Custom** with substitution variables, ensure that the validation script expression evaluates to an array of strings.

The length of the array must be equal to the number of substitution variables in the message. See Example 2 Custom Validation Message with Substitution Variables for details.

10. Confirm that the **Enabled** check box is selected, and click **Finish** to complete the process of defining a validation.

Editing a Validation

Editing a validation involves similar steps as adding a validation.

Procedure

1. With the form open in the Form Designer, select the control whose validation you wish to edit or delete.

2. In the Properties view of the control or the pane, click the **Validations** tab.

3. From the **Validations** tab of the Properties view, edit the **Name**, **Execute When**, **Message Type**, **Message**, and **List** fields.

> If the message is an external reference, a cell editor appears on clicking in the message cell. Clicking the cell editor opens the **Resource Picker**, from where you can select an appropriate message key.

4. Select the script you wish to edit.

   An ellipsis **(...)** button appears next to the script.

5. Click the ellipsis **(...)** button.

   The Edit Validation Script dialog opens.

   The Edit Validation Script Dialog
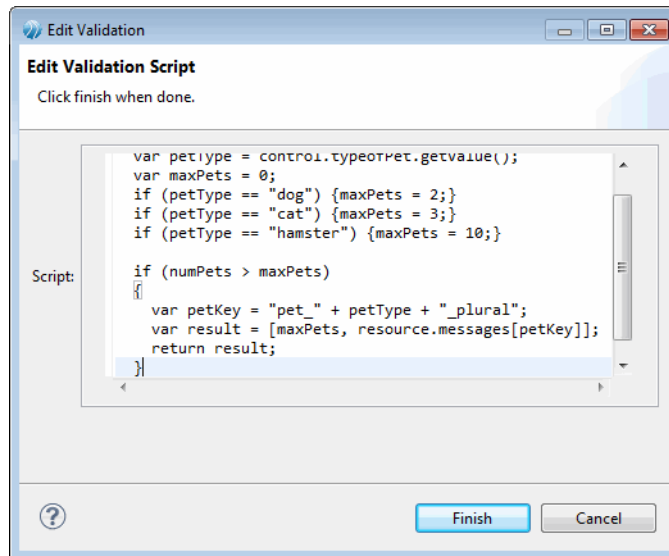
6. Edit the code in the **Script** field, and click **Finish**.

Result

> The script editor provides content-assist editing. On typing the beginning of a legal value, such as "`control.`", a pop-up window appears listing the available completion proposals. If you type **CTRL +Space**, a list displays containing all the top-level variables that are available in the given context.

Examples of Validation Messages

You can configure validation messages from external references, or you can also specify a custom validation message.

Example 1 Setting a Custom Validation Message

In this example, the text field has the name **petNumber**.

The General Tab

This means that the value submitted for this text field by a user can be referenced in the validation script by the expression `control.petNumber.getvalue()`.

Procedure

1. In the Define Validation dialog, specify the value in the **Script** field.

Defining Custom Validation

```
Define Validation

Define Validation
  Click finish when done.

Name:          valid_pets_custom
Execute When:  ○ On Form Submit  ◉ On Value Change
Validate As List: ☐

               var numPets = context.value;
               var petType = control.typeofPet.getValue();
               var maxPets = 0;
               if (petType == "dog") {maxPets = 2;}
               if (petType == "cat") {maxPets = 3;}
               if (petType == "hamster") {maxPets = 10;}

Script:        if (numPets > maxPets)
               {
                   var petKey = "pet_" + petType + "_plural";
                   var result = [maxPets, resource.messages[petKey]];
                   return result;
               }
               return true;

               ○ External Reference:  [                    ] [...]
Message:
               ◉ Custom:             [The number exceeds the number of allowed pets]

Enabled        ☑

   (?)                                         [ Finish ]  [ Cancel ]
```

2. In the **Custom** text field, type the validation error message that you want the user to see on specifying incorrect data.

3. Confirm that the **Enabled** check box is selected, and click **Finish**.

   If the user submits a value other than the one specified in the validation script, the validation error message appears on the form.

   Validation Script Example 1

   ⚠ The number exceeds the number of allowed pets

Example 2 Custom Validation Message with Substitution Variables

You can specify a validation message with substitution variables.

Procedure

1. In the Define Validation dialog, type the code as shown in the **Script** field in the figure Defining Custom Validation Using Substitution Variables.

TIBCO Business Studio™ Forms User's Guide

Defining Custom Validation Using Substitution Variables
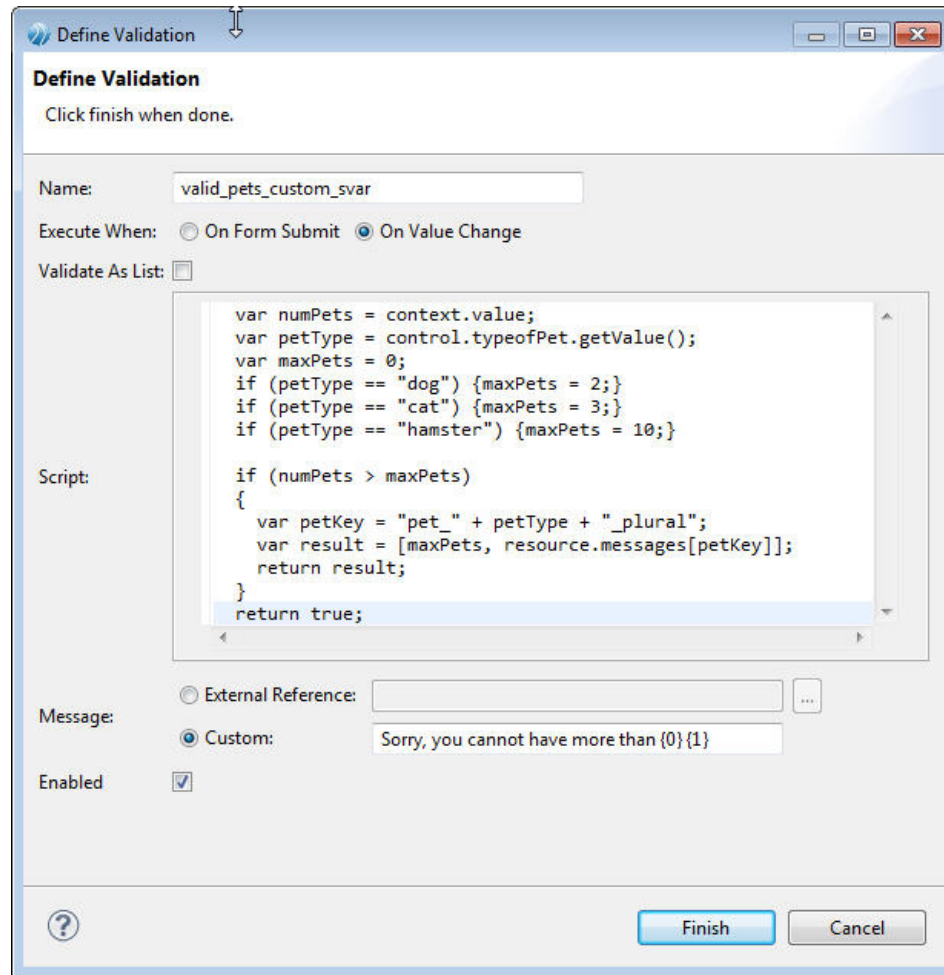
2. In the **Custom** message field, specify the validation error message using substitution variables from an array.

3. Confirm that the **Enabled** check box is selected, and click **Finish**.

   If the user submits a value other than the one specified in the validation script, a validation error message using the substitution variables from the array appears on the form.

   Validation Script Example 2

   ⚠ **Sorry, you cannot have more than 3 cats**

Example 3 Validation Message Referenced from External Resource

You can specify a validation message from an external resource.

Procedure

1. Create `<validations>.properties` file under the **Presentation Resources** special folder in **Project Explorer**.

   The name of the file does not matter as long as the extension is `.properties`. The file can contain any arbitrary custom display strings, not necessarily only validation messages.

2. Define validation messages in the `<validations>.properties` file.

   The validation message key must have "`validation_`" as a prefix. If a key does not start with "`validation_`", the system does not treat it as a validation message.

3. Add the newly-created `<validations>.properties` file to the resources list of the form.

   After adding the `.properties` file as a form external resource reference, the new validation messages are available in the Resource Picker.

4. In the Define Validation dialog, provide the details of the external resource reference.

   The Define Validation Dialog Using External Resources

5. Click **Finish**.

   If the user submits a value other than the one specified in the validation script, the validation error message from the external resource file appears on the form.

   > You can localize the validation error messages. See Localization of Forms .

Calling External JavaScript Functions

Often, a single JavaScript function is useful for many different forms. Typical utility functions are reused commonly, such as functions for validating common types of input, or for making calls to external services, and so on.

It is not necessary to rewrite or copy these functions from one form to another. To facilitate reuse, common JavaScript can be placed in one or more JavaScript files external to the form. These JavaScript files are deployed to the WebDAV server with your form files, and can be used by multiple forms in the browser client.

To use an external JavaScript file in a form, you need to add it to the form resources. Once added, the JavaScript files get deployed automatically when the form is deployed, and loaded at runtime before the form is loaded.

Specialized Layouts

You may need to resize, re-sequence, and nest panes to create forms with specialized layouts.

Nesting Panes

Panes may be nested within other panes to achieve specialized layouts. In particular, panes with different layout directions can be nested to achieve column- or row-wise layouts.

You cannot add a modal dialog pane to another modal dialog pane, nor to a record pane, nor as a direct child (tab) of a tabbed pane.

Creating Columns with Nested Panes

You can create a multi-column layout by nesting two vertical panes, side-by-side, within a horizontal parent pane.

Procedure

1. Place groups of controls into two separate vertical panes, each representing a separate column.

2. Drag the second pane to a position next to the first pane, so that you see a dotted line appear. The dotted line means that a horizontal pane will be automatically created for you to hold the two vertical panes.

   As you drag the pane, you will see feedback on the new position of the pane prior to releasing the mouse button.

3. If you want more than two vertical columns, drag additional panes, one at a time, next to the right-most vertical pane within the new horizontal parent pane.

Positioning Controls into a Multi-Column Layout

A multi-column layout is created by positioning multiple vertical panes within a horizontal pane. The creation of a two-column layout is used here to demonstrate this technique.

Procedure

1. Vertical panes A, B, C, and D are placed on the form, one beneath the other.

   Place Vertical Panes on the Form

2. Drag the **Pane B** up and to the right, close enough so that a colored background appears around **Pane A**.

Position the New Pane
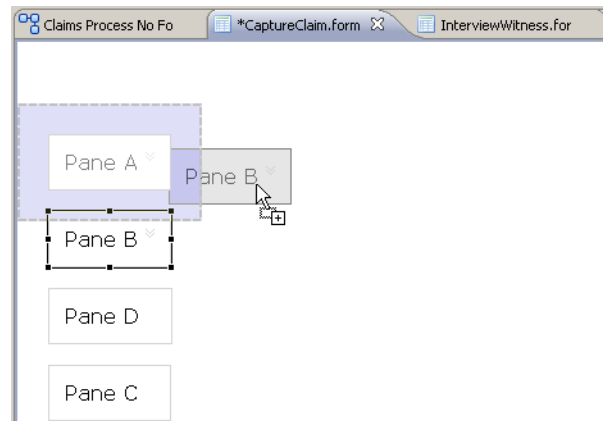
3. A new horizontal pane is automatically created, containing the two vertical panes, side by side.

New Horizontal Pane is Automatically Created
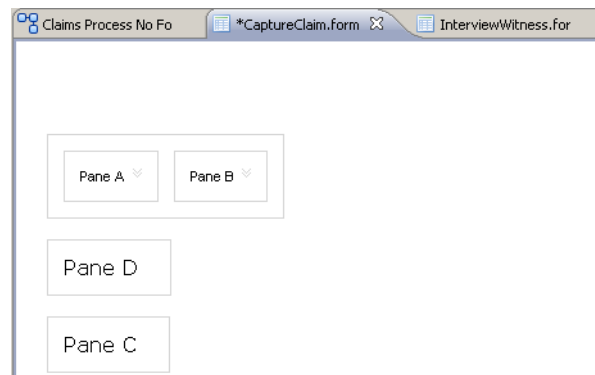
Resequencing Tabbed Panes

Tabbed panes can also be resequenced in the Outline view using drag-and-drop.

Procedure

1. Expand the tabbed pane using the arrow to the right of the tabs.
2. Use drag-and-drop to move the child pane to its new position.

Result

The pane's tab will automatically adjust itself to the new index position.

Resizing a Tabbed Pane

If you add or delete child panes within the tabbed pane, or add or remove controls from a child pane, or move controls between panes, you may need to resize the tabbed pane to account for the resulting increase or decrease in the child pane's width, height, or both.

Procedure

1. Collapse the tabbed pane.
2. Select each tab in turn, checking that all child panes fit comfortably within the tabbed pane's content area and resize as necessary.
3. Verify the run-time appearance by clicking the **GWT Preview** tab in the Form Designer. If scrollbars appear or there is excessive unused space, you may need to make further adjustments.

Positioning a Modal Dialog Pane

By default, a modal dialog pane is configured to render at the center of the window. However, you can change its position to suit the form.

Procedure

1. Select a modal dialog pane, and display its tab from the **Properties** view.
2. Specify the **Dialog Position**.

   The available choices are:

Option	Description
Center of the window	Default. Use this if you want the modal dialog pane to be displayed at the center of the window.
Center of the form	Use this if you want the modal dialog pane to be displayed over the form, when the form is being rendered as a part of a larger application.
Relative to the focused element	Use this if you want the users to fill the data in the modal dialog pane with a context to the focused element. (0,0) is the top left position of the element clicked by the user. Specify the X and Y coordinates accordingly. Positive values move the pane downward and to the right by specified pixels.
Absolute Position	Use this if you want the modal dialog pane to be displayed at an absolute position irrespective of the scrolling. (0,0) is the top left point in the view port. Specify the X and Y coordinates accordingly. Positive values move the pane downward and to the right by specified pixels.

3. Click **Save**.

Setting Visibility of Pane and Control Borders

The controls and panes on a form, including nested panes, are sometimes clearer and easier to distinguish from one another when viewed with borders around them. The borders do not appear at runtime (or in the GWT Preview mode), but only in Design mode.

It is a matter of personal preference whether to display the borders in Design mode. To switch between showing and hiding borders around controls and panes, click the **Toggle Pane and Control Borders** button at the far right of the TIBCO Business Studio Forms toolbar.

Procedure

1. Click **Window** > **Preferences** to open the Preferences dialog.
2. Click **Form Designer** in the left navigation pane.
3. Select or clear the **Show pane and control borders** check box as desired, and click **OK**.

   Borders are displayed or hidden as specified.

Embedded Forms

Many forms may use similar sections, such as profile information of a user, contact information, postal address, and so on. You can create these sections separately, and reuse them in several forms.

To reuse such sections, you need to create a reusable fragment of a form separately, and embed it later in parent forms. These reusable sections are called Embeddable Forms.

In the Embedded Forms topic the following terms are used frequently:

Embeddable Form A form that has been designed to be embedded is referred to as an embeddable form.

Embedded Form Once a form is embedded within the parent form, it is referred to as an embedded form.

For example: you have to design a form for delivery of goods to customers. In such a form, different types of address information is required, such as delivery address and personal address. If you design a normal form, you have to create the same set of address fields at two places. By using the embedded forms feature, you can create a reusable embeddable form with the address fields and embed this form at multiple locations in the parent form.

Prerequisites of an Embeddable Form

An embeddable form has no navigation or message panes, as navigation and messaging are taken care of by the parent form.

If you want to embed an existing form within another form, it is advisable to make the following changes to make the existing form suitable for embedding:

- Remove the navigation and messages panes from the embeddable form.

- If the embeddable form has any dynamic behavior that must be exposed to the parent form, you must tie the dynamic behavior to parameters on the embedded form, which can then be updated by parent forms.

Working with Embedded Forms

When you create an embeddable form, you also need to create different gestures for embedding that form.

Creating an Embeddable Form

You can create an embeddable form from the Project Explorer.

Procedure

1. Go to the **Forms** folder, or any folder under the Forms folder in the Project Explorer and click **Context Menu** > **New** > **Form** . The New Form dialog opens.
2. On the New Form dialog box, specify the **File name**. Select the **Form type** as **Embeddable**.
3. Click OK.

Result

This newly created form will only have a single root pane. Messages and navigation panes are not created.

Embedding a Form by Using the Embedded Form Icon

The embedded form icon ![icon] is displayed on the Palette in the Panes section.

Procedure

1. Select the embedded form icon from the Palette and drop it in the required location on the Form Designer canvas.

2. The Select the form to embed dialog is displayed. All the forms available in all the projects in the workspace are listed in the dialog. Select the required form.

> If the selected form is from another project, you are prompted to add the other project as a reference.

3. The Embedded Form dialog is displayed asking you to map the embedded form parameters in the 'Mapping' property section. Click **Yes** to continue or **No** to skip the parameter binding. See Embedded Form Parameters for the details of parameter binding.

Embedding a Form from the Project Explorer

You can embed any form within the project or any project on which the existing project depends.

Procedure

1. Select the form from the Project Explorer and drop it in the required location on the Form Designer canvas or Outline view.

2. The form is embedded within the form. An embedded form is represented as a pane containing a form icon, labelled with the name of the embedded form.

> A form is embedded only at design time. You can have multiple levels of nesting. The nested form is embedded by reference.

Adding a BOM Class or Form Parameter to a Form

You can create an embeddable form UI components directly from a BOM class.

Write the task in procedure format post conversion.

Select a BOM class in the Project Explorer and drop it in the Form Designer canvas. All the UI components associated with the BOM class are automatically created on the form.

Similarly, you can select a form parameter in the Outline view and drop it in the Form Designer canvas. This will also automatically create all the UI components associated with the parameter.

> It is recommended to define a separate project with all the reusable embeddable forms along with the BOM classes they represent. Add this project as a dependency in other projects to make use of the data model.

Embedded Form Parameters

Once a form is embedded within a parent form, the embedded form parameters can be accessed only via the parent form. An embedded form exposes an interface that consists of its parameters.

The panes and controls in an embedded form are generally bound or otherwise mapped to its parameters. These parameters in the embedded form are in turn mapped to parameters, data fields, controls, or panes in the parent form.

For example: we have an embeddable form which contains a single pane that is bound to a parameter of particular type defined as a BOM class. This form is embedded in a parent form. You bind an embedded form parameter to one of the parent form's **IN OUT** parameters of the same type. When the parent form is loaded with an instance of that parameter, the embedded form is updated via the binding. This is one of the mechanisms by which information is exchanged between the parent form and the embedded form

There are many ways in which data can be exchanged between the parent and the embedded forms:

- Using absolute bindings from parent form panes or parameters
- Using computation actions

- Using the API in script actions

For details of how to set bindings and actions, see Setting Bindings and Setting Actions.

Accessing Embedded Form Parameters

You can access the embedded form parameters using action scripts and computation actions.

The parameters of the embedded form appear as Data Fields in the deployed copy of the parent form. The names of these parameters are scoped by the name of the embedded form.

Example:

```
data.get<EmbeddedFormName>_<ParamName>();
```

For example, `data.getCustomerForm_Customer();`

Setting Bindings from the Mappings Tab

You can bind parameters of a parent form to an embedded form.

Procedure

1. In the Form Designer canvas or Outline view, select the embedded form.

2. Go to the **Mappings** tab in the **Properties** view of the parent form.

3. All the parent form parameters are displayed in the left pane. The right pane displays each embedded form, along with the parameters defined in that embedded form.

4. Drag the required parent form parameter and drop it onto the embedded form parameter to bind it. This creates the required binding, which is represented by a connecting line between the parameters.

Rendering of Embedded Forms

On the Form Designer canvas, an embedded form is represented as a pane containing a form icon. When the builder runs, it creates a deployable copy of the parent form. Each embedded form pane is replaced by the contents of its respective embeddable form, recursively.

At preview and runtime, the GWT implementation renders the deployable copy of the parent form.

Preview Rendering of the Parent Form

Editing Embedded Forms

You cannot directly edit an embedded form within the context of the parent form. It is possible to move it to a different location within the form, but it cannot be edited directly.

Procedure

1. In the Form Designer canvas or Outline view, select the embedded form pane.

2. Go to the **Properties** tab in the **Properties** view of the embedded form pane. The Form Reference displays a link to the embedded form.

3. Click the link to open the embedded form in the Form Designer.

4. Click the ellipsis **(...)** button to change the embedded form.

5. Update the embedded form using the Form Designer.

   The updates are available in the parent forms without having to re-embed the form.

   > The changes made in the embedded form can be seen in preview and at runtime after the parent form is redeployed.

Mappings Tab

The **Mappings** tab of the Properties view provides a global view of all the bindings and computation actions related to the selected element in the Form Designer canvas or Outline view.

You can view, edit, and create mappings from the **Mappings** tab. It displays the values of the source and target fields of the existing mappings in the left and right trees respectively.

> The term *mappings* used in this topic, is a generic word which covers both bindings and computation actions.

Mappings Tab of the Properties View
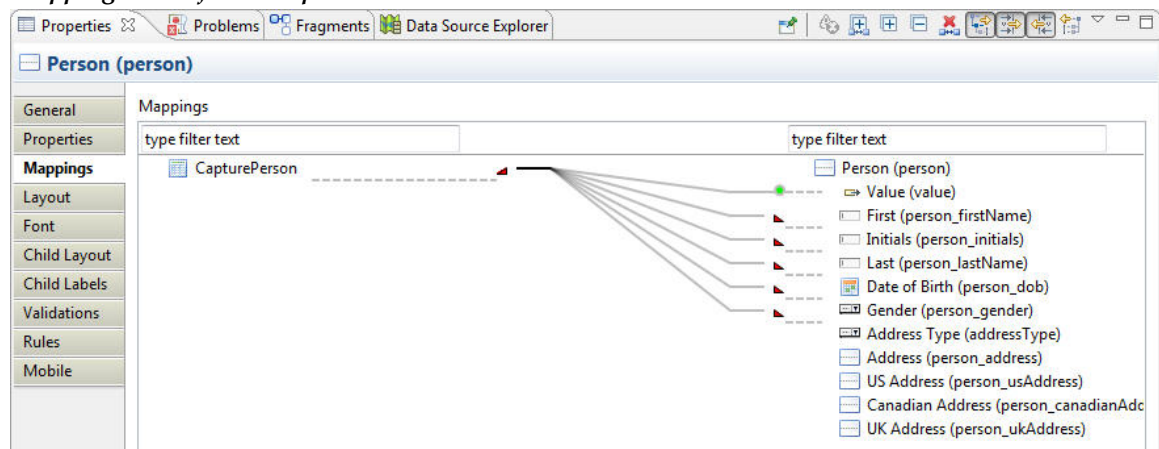
The details are as follows:

* The right pane displays the bindable properties of the selected target element.

* The left pane displays the bindable source properties to which the target elements are bound. It displays the selected object and its ancestors all the way up to the containing form and also includes the form parameters and data fields.

* The connecting lines represent the existing mappings between the source and target properties.

* The arrow end-point represents unidirectional mappings.

* The red triangle at one end-point of the connecting line represents collapsed mappings.

The default view of the **Mappings** tab is focussed on the mappings of the selected element.

A set of buttons and filters are provided in the toolbar. Each of these filters has a corresponding toolbar button and a toolbar menu item: both are associated with same filter action. The buttons control the depth to which the source and target trees are expanded. The filters help you to control the properties

to be displayed in the source and target panes. The details are explained in the table Toolbar Buttons for the Mappings Tab.

Toolbar Buttons for the Mappings Tab

Button	Description
	Expands the source and target trees just to the extent required to reveal all the existing mappings.
	Expands both the source and target trees to the maximum possible extent.
	Collapses both the source and target trees to the maximum possible extent.
	Deletes all the bindings and computation actions related to the selected element.
	This filter shows only the selected element and its related ancestors in the source tree. By default this filter is enabled. When disabled, unrelated components are also visible but initially shown collapsed. You can expand these unrelated nodes manually.
	This filter shows only the bindable value property in the source tree. By default this filter is enabled. When disabled, the other bindable properties of the selected element are also displayed in the source tree.
	This filter shows only the bindable value property in the target tree. By default this filter is enabled. When disabled, the other bindable properties of the selected element are also displayed in the target tree.
	This filter hides the descendants of the selected pane in the target tree. By default this filter is enabled. When disabled, all the target pane's children are visible but initially shown collapsed. You can expand the child nodes manually.

Coloration Feedback

The connecting lines representing the existing mapping can be difficult to understand especially if there are many mappings between the elements of the source and target tree. The coloration feedback is very useful in such scenario as it allows you to see at a glance which mappings are defined within a given component tree.

Some examples are:

- When you select a bindable element in the source or target tree, all mappings involving that element and its visible children are highlighted in bold. In the figure Mappings Tab of the Properties View, when you select `Customer_order_item_SKU/Value` node in the target tree, the corresponding binding is highlighted in bold. This is especially helpful when the 'show only source ancestors' and 'hide target descendants' filters are disabled.
- When you click a collapsed mapping (represented by a red triangle), it automatically expands and displays both of the end-points of the mapping.

The **Mappings** tab's user interface (UI) simplifies tasks such as property binding and creating computation actions.

Setting Bindings

Bindings are represented by a connecting line between the properties.

Procedure

1. In the Form Designer canvas or Outline view, select the target element.

2. Go to the **Mappings** tab in the Properties view of the selected element.

   > If you are adding a binding to a grid pane control, make sure the **Visible** property is enabled.

3. Drag the property of a component, parameter or data field from the source tree and drop it over the property of a component in the target tree to which you want to bind it.

   You can also create a binding in the opposite direction, that is from the target tree to the source tree.

Adding Computation Actions

You can add a computation action from the mappings tab of an element.

Procedure

1. In the Form Designer canvas or Outline view, select the component or element for which you want to add a computation action.

2. Go to the **Mappings** tab in the Properties view of the selected element.

   > If you are adding a computation action to a grid pane control, make sure the **Visible** property is enabled.

3. Click the New Computation Action node in the source tree. By clicking on this node, you can specify the name of the computation action in direct edit mode.

4. Press Enter to commit the newly created computation action name and display the **Rule Details** page of the New Rule dialog.

5. Follow the instructions given in the Adding a Rule Using the Outline View section to create a new computation action.

   After the new computation action is created, it is visible in the source tree.

6. To connect the newly-created computation action to its destination, drag the computation action and drop it on the target property of a component in the target tree.

   This completes the creation of computation action.

Editing Computation Action Using the Script Editor Section

You can update all the fields of the computation action from the Script Editor.

Procedure

1. Go to the **Mappings** tab in the Properties view.

2. Select the computation action to be edited, from the source tree.

3. Expand the Script Editor section to see the computation action section in the **Mappings** tab view, and update all the fields of the action.

Editing Mappings

You can edit the binding or the computation action from the Mappings tab.
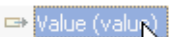
Procedure

1. In the Form Designer canvas or Outline view, select the target element.

2. Go to the **Mappings** tab in the Properties view of the selected element.

3. Select the mapping to be edited and invoke the **Edit Binding** or Edit Computation Action dialog by:
   a) Double-clicking the selected mapping
   b) OR presssing Enter
   c) OR executing **Context Menu** > **Edit**

   The **Edit Binding** or Edit Computation Action dialog is displayed. See Setting Bindings and Setting Actions for details.

   > When you create mappings from the **Mappings** tab, it prevents you from creating invalid mappings. Look for the following cursor feedback when you drag a property between trees:
   >
   > - ⇨ **(value)** - Not valid binding.
   >
   > - ⇨ **Value (value)** - Valid binding

Property Resource Bundles

In TIBCO Forms, you can configure the resource keys in the Property Resource Bundles or `.properties` files. You can override the values of the existing resource keys, and also add new resource keys.

Such customizations may be necessary for:

- Changing the value of a resource key, for example the default date format used by all the controls

- Adding a new locale for adding a new language that is not already listed in the default locales

- Adding new resource keys, for example new numeric formats

- Adding a new `.properties` file that is automatically added to all the forms in a project, or to all the projects in a workspace

- Using implicit validations that use the messages specified in the common resource bundle

It is possible to do such customizations at the project level and also at the workspace level.

For information on the default common resources, see Common Resource Keys.

The Merging Process

TIBCO Forms creates a merged bundle of common resources from the overridden resource keys and the default resource keys from the base bundle. This merged bundle resides in the Presentation Resources folder.

> The `.common` sub-folder is hidden by default. To display it, you need to disable the `.*resources` Project Explorer filter.

The entries in your `<custom>.properties` file are compared with the existing entries in the default `common.properties` file. If a resource key already exists in the default file, its value in the `<custom>.properties` file is used in the merged bundle. If the resource key is not in the default `common.properties` file, it is added to the new merged file.

If the custom bundle does not specify a file for a specific locale, the entire file from the default bundle is passed on to the merged bundle. Similarly, you can also specify a new locale that is not a part of the default bundle.

The figure Merging Process illustrates the merging process.

Merging Process
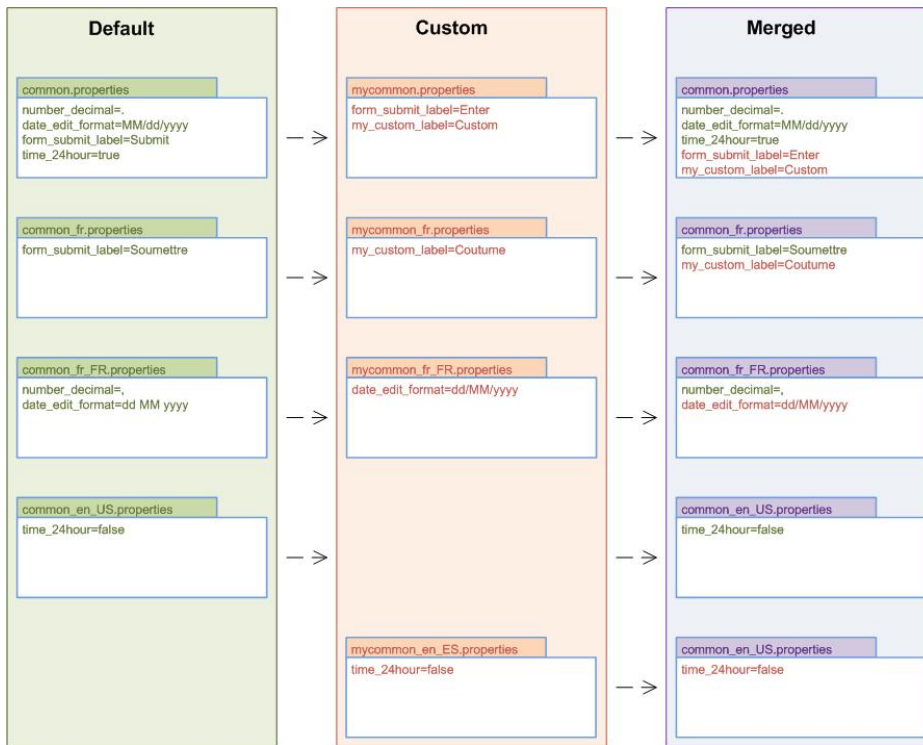
Customizing Property Resource Bundles

You can override the values of the properties in the resource bundle at the project level, or at the workspace level.

Procedure

1. Right click the **Presentation Resources** folder and click **New** > **File** .

   The New File dialog appears.

2. In the New File dialog, name the file with `.properties` as its extension, and click **Finish**.

   In the example, the name of the file is *customer_specific* and the extension is `.properties`.

3. In the new `.properties` file, type the resource entries that you wish to add or override.

   Sample Resource Entries

   In this example, the new resource key is:

- `validation_PetCount`

  The table Example Resource Keys with Overridden Values lists the existing resource keys with their default values and their new values.

  Example Resource Keys with Overridden Values

Resource Key	Default Value	Overridden Value
`dpane_new_label`	Add a new record	Add a new report
`time_24hour`	true	false
`form_submit_label`	Submit	Enter

  The merged common resources bundle now consists of the old resource keys with the new overridden values along with the new resource keys.

4. In the Project Explorer, right click the project, and select **Properties**.

   OR

   Click the **Project** menu, and select **Properties**.

   The Properties for project name dialog opens.

5. In the left pane, click the **Form Designer** arrow to expand it, and select **resources** > **Common properties**.

   > You can also specify the properties file at the workspace level from this dialog. To do that, click the **Configure Workspace Settings** link. When opened this way, the dialog shows filtered options, and it only shows the **Form Designer** and **Common properties** file.
   >
   > Else, you can go to **Window** > **Preferences** , and expand the **Form Designer** to select **resources** > **Common properties**. Continue the remaining procedure from the next step.

6. Select the **Enable project specific settings** check box.

7. Click the **Browse** button next to the **Common properties File**.

   The Pick Resource dialog opens.

8. Select the new properties file, and click **OK**.

9. Click **Apply**, and in the ensuing Rebuild? dialog, click **Yes**.

   In the **Properties** view > **Resources** tab, the URI field shows that the common properties resource is now overridden.

Validations Related to Custom Common Resources

The default validations available on custom common resources have the following objectives:

- To check if the project has a project reference to the project containing the common properties override
- To check if an override is set at the project level or workspace level
- To check if the properties override file actually exists
- To warn about any form that uses a form-level common properties override

If you see such validation messages, do one of the following as appropriate:

- Add the missing project reference

- Create the missing `common.properties` file
- Remove the common properties override from the preference node
- Remove the common properties override from the form

Customizing the Form's Preview Data

By default, when a form is previewed, sample data is included for each control to give a better idea of how the form appears to a user at runtime. You can customize the preview data that appears, rather than using the default data generated for each control type.

Editing the File form-name .data.json

The `.data.json` file is a generated file. Do not edit the original file. If you edit it, your customizations are overwritten when the file is regenerated.

Also, be sure to maintain the file extension, `.data.json`. Otherwise, your customized preview data file is not accessible to the form.

Procedure

1. In the Project Explorer, find the file that contains the preview data.

   The default location of this file is: [*project-name*] > **Forms** > **ProcessPackage** > [*business-process-name*] > [*user-task-name*] > [*form-name*].data.json

2. Right-click the `.data.json` file and click **Copy**. Then, in the same location in the Project Explorer, right-click and click **Paste**.

   The Name Conflict dialog appears asking you to type a new name for the file. Rename the file keeping the extension `.data.json`. Do not delete the original preview data file.

3. Right-click your newly-named custom preview data file, and click **Open With** > **Text Editor** .

4. Edit the file, providing your desired values for the preview data in place of the default values in the file.

Result

Example of Default Preview Data File

```
{ items: [
   { $param:'AnotherDemo', $value:
{$type:'com.example.demo.Demo',                 normalText:
"normalText"    ,list:[
   "list"
   ],duration:         ""     ,attribute1:[
   "2010-05-16"
   ]}
   },
   { $param:'Demo', $value:
{$type:'com.example.demo.Demo',                 normalText:
"normalText"    ,list:[
   "list"
   ],duration:         ""     ,attribute1:[
   "2010-05-16"
   ]}
   }
   ]}
```

Example of Customized Preview Data File

```
{ items: [
   { $param:'AnotherDemo', $value: {$type:'com.example.demo.Demo',
   normalText: "My Sample Data"    ,
   list:["list", "John", "George", "Ringo"],
   duration: "P4Y"      ,
```

```
    attribute1:["2010-05-07", "2010-02-11"]}
  },
  { $param:'Demo', $value: {$type:'com.example.demo.Demo',
    normalText: "normalText", list:["list"],duration: "",
    attribute1:["2010-05-07"]}
  }
]}
```

Configuring the Setting in the Properties View

Once you have created a custom preview data file, you can configure the form to use this file rather than the default file (or no file at all) for preview data.

Procedure

1. In the Properties view for the form, click the Preview Data tab.

2. Select one of the following radio buttons:

   None

   Select this option if you prefer that no data be displayed initially for the controls when the form is previewed.

   Default

   Select this option if you want to use the default data for each control on the form.

   Custom

   Select this option if you want to use your customized `.data.json` file for the preview data values. The **Custom** radio button is paired with an optionlist that shows all the `.data.json` files associated with the current form. Select the custom preview data file you want to use from the optionlist.

Form Data Fields

Form data fields are used to store data that is needed only for the lifetime of the form.

User task parameters offer a way to associate a user task with process data fields so that data that is available to the entire process can be used, viewed, or modified through the form associated with the user task. But in some cases, you want to track data that is useful for the functioning of the form, but is unrelated to other tasks in the process and is not needed by the server. In such a case, instead of using parameters, you can create one or more *form data fields* to store that data for the lifetime of the form.

The same data types available for parameters are also available for form data fields. The key difference between a form data field and a parameter is that a form data field has no **Mode** property (In, Out, or In/Out). Since a parameter's **Mode** property is used to specify the way parameter data interacts with the larger business process, it has no relevance to form data fields.

Configuring a Form Data Field

You can use a form data field to make a set of invisible panes in the form visible when user specifies a certain value (or takes other action). In this case, by using a form data field you can track which of those panes are visible. The form data field functions as a global variable within the context of the form.

You can also use a form data field in a form containing a wizard pane to track which page of the wizard is currently visible to the user.

Procedure

1. Open the form in the Form Editor view, if it is not already open.

2. In the **Outline** view for the form, right click the **Data** folder and click **New Data Field**.

3. Provide a label, name, and type for the data field.

4. Select **External Reference** to choose a type from all the types defined for the process.

Numeric Controls

A numeric control is not a distinct control type, but is a special property that can be enabled for a text input control. It is used to display data in a specified format so that it is easier to read.

The numeric control property of a text control enables you to specify the display format of numeric and currency values. It only changes the way the control value is displayed and does not affect the way the value is edited or saved.

To define a format, you can use the following pattern:

```
PosPrefix PosFormat PosSuffix;NegPrefix NegFormat NegSuffix
```

The spaces between prefix, format, and suffix are used only for clarity and should not be included in the actual format.

This pattern defines a format for positive numbers (`PosPrefix PosFormat PosSuffix`) and a format for negative numbers (`NegPrefix NegFormat NegSuffix`) separated by a semicolon (;).

The format can include the formatting characters shown in the table Numeric Control Formatting Characters. Each character is replaced with locale-specific text when the number is formatted.

Numeric Control Formatting Characters

Character	Description
0 (Digit)	Used to signify the minimum number of digits to be displayed. Each instance of the character represents a position for one digit. If no value exists in a position, a zero (0) is displayed. This character is not valid within prefix or suffix. Left of the decimal point: leading 0's are shown. Right of the decimal point: trailing 0's are shown.
# (Optional Digit)	Used to signify the minimum number of digits to be displayed. Each instance of the character represents a position for one digit. If no value exists in a position, a blank space is displayed. This character is not valid within prefix or suffix. Left of the decimal point: leading 0's are not shown. Right of the decimal point: trailing 0's are not shown
. (Decimal separator)	Used as a numeric or monetary decimal separator. This character is not valid within prefix or suffix and is localized based on the locale settings.
- (Minus sign)	Used to indicate a negative number. This character is only valid in the prefix or suffix.
, (Grouping separator)	Used to group the number format. The grouping separator must not be used to the right of the decimal point in a number format. This character is localized and is not valid within prefix or suffix.
;	Separates positive and negative sub-patterns. This character is not valid within number format, prefix or suffix.

Character	Description
¤	Currency sign (Unicode code point-\u00A4). This character is valid only within prefix or suffix and is replaced by the localized currency symbol.

Some sample formats are listed in the table Numeric Control Sample Formats:

Numeric Control Sample Formats

Number	Format Pattern	Displayed
0	0	0
0	#	
123	0	123
1234.123	#,###.0000	1,234.1230
1234.123	#,###.00	1,234.12
1234567.123	#,###.00	1,234,567.12
1234.123	000,000.00	001,234.12
1234.12345	#,##0.00##	1,234.1234
1234.123	#,##0.00##	1,234.123

Inserting a Numeric Control

You can specify the display format of a numeric control either using an external reference or by using your own custom format.

Procedure

1. Select a text input control from the Palette and drop it in the form.
2. Go to the **Properties** tab in the **Properties** view for the text input control and select the **Numeric** check box. This enables the **Format** options.
3. Specify the display format from the following options:
   a) E**xternal Reference**: Select a format from an external resource. See Inserting External Reference Format
   b) **Custom**: Define a custom format. See Inserting a Custom Format.

Inserting External Reference Format

By selecting the **External Reference** option, you can use one of the predefined formats from the common resource bundle.

Procedure

1. Select **External Reference** under the **Format** options.

2. Click the ellipsis **(...)** button to display the **Resource Picker**.

3. Select a format from the list and click **OK**.

Creating a Custom Format

You can also create your own custom formats and add them to the **Resource Picker** list. The new custom formats must be placed under the **Presentation Resources** special folder.

A sample custom format is as follows:

```
format_myformat1 = 000.000
  format_myformat2 = \u00A4#,#0.0;[\u00A4#,##0.0]
```

'\u00A4' is the Unicode value for the ¤ currency symbol.

Procedure

1. In the Project Explorer, go to the context menu of the **Presentation Resources** folder and click **New > File.**

2. On the New File dialog box, type the file name and use the extension `.properties`. The builder creates matching `<name>.properties.json` and `<name>.locales.json` files in the same folder.

3. The newly-created properties file is automatically opened in the Properties File Editor for editing. Edit the file to add custom number formats.

Adding the Properties File in the Resource List

The newly-created properties file must be added to the resources list of the form.

Procedure

1. Go to the **Resources** tab in the **Properties** view at the root level of the form.

2. The common resource bundle (common) and the default resource bundle for each form (form) are predefined for each project.

3. Click the **(plus)** button to display the Pick Resource dialog box.

4. Select the newly-created `.properties` file from the list, and click **OK**.

   Once you have added the newly-created `.properties` file as a form external resource reference, the new formats are available in the **Resource Picker**.

Inserting a Custom Format

By using the **Custom** option, you can choose from some example formats or define your own format inline.

Procedure

1. Select **Custom** under the **Format** options.

2. Type the custom format in the text box. A list of example formats is available in the selection list.

3. You can select one of the example formats or define your own format inline using the formatting characters listed in the table Numeric Control Formatting Characters.

Editing a Numeric Control

To edit a numeric control, the text input control must have focus.

For editing, the number is displayed in the raw format and in full precision. The prefix, suffix, and the group separators are not displayed. The decimal point is displayed using the conventions of the active locale. You can edit the values and move out of the control.

When the text input control loses focus, the value in the text input control is displayed using the specified display format.

Localization of Forms

With TIBCO Forms, you can create forms that support multiple languages.

Form logic, including layout and control types and validation rules, is stored in the form file. Language-specific information, including labels and validation messages, is stored in locale-specific properties files.

To simplify the localization of forms, all text that appears on a form is stored in a properties file. The properties file includes the strings that make up the labels for controls and panes, as well as the strings for hints, validation messages, and error messages.

You can view the localized version and change the locale of a form in the preview pane.

Each form has a *base* properties file that is generated automatically when the form is created, and is regenerated each time the form is saved. This file appears in **Presentation Resources** special folder in the Project Explorer. The base properties file contains the strings that appear as labels and messages in the form's property sheets.

Base Properties File
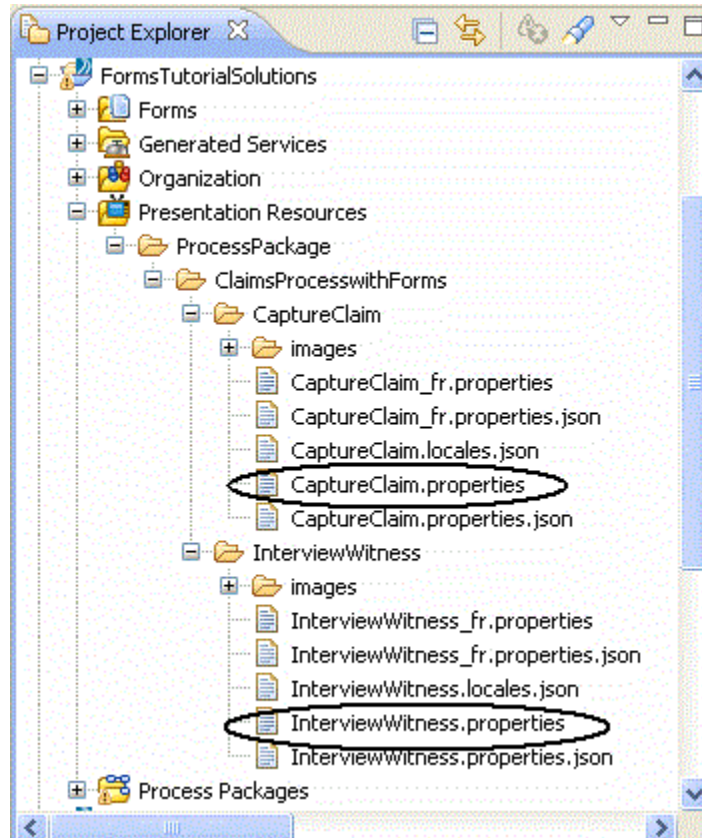
To create a localized version of a form, you will make a copy of the base properties file, rename it, and edit the strings it contains.

⚠️ *Do not edit the strings in the base properties file itself.* Any changes you make to this file will be lost as soon as the project is built (which is to say, as soon as you save the form, with the default setting, where *auto-build* is enabled). To change the labels and messages for the base version of the form, use the form's property sheets instead. The changes you make in the property sheets will appear in the base properties file when the form is saved.

The renamed locale-specific versions of the properties file will *not* be automatically regenerated, and thus your locale-specific strings will not be lost when the form is saved.

Creating a Locale-specific Properties File

The localized version must be present in the same directory, which contains the original base properties file.

Procedure

1. Select the `<form>.properties` file from the **Presentation Resources** special folder in the Project Explorer. Make a copy of this file for each locale.

2. Rename the copy, using the naming conventions for languages and regions. See Language-specific and Country-specific Properties Files for more details.

3. For every new properties file created in the **Presentation Resources** folder, the builder automatically creates a matching <file.`properties.json` file at the same location.

4. Open a locale-specific version of the properties file in the Properties File editor and manually translate the strings into the desired language.

5. Click **Project** > **Clean** to clean and rebuild the project. This updates the `<form>.locales.json` file with the details of the language in which the form has been localized. For example, if you create `DemoForm_fr.properties` file, then the `Demoform.locales.json` will contain `["fr"]`. This file is updated when the you rebuild the project after creating a new locale-specific version of the properties file.

6. Run the JDK command-line tool **native2ascii**, using the locale-specific properties file as input, to ensure that the file contains only ISO_8859-1-encoded characters:

```
C:\BusinessStudioWorkspace\Forms>native2ascii LocDemo_fr.properties_
```

The native2ascii command-line tool is available in the directory `%JDK_HOME%\bin`.

7. Move the completed locale-specific version or versions into the same directory where you found the original base properties file.

   💡 You can find the directory that contains all the properties files by using the context menu of one of the form's files in the Project Explorer (for instance, the base properties file) and clicking **Properties** to open the properties dialog. The path to the selected form resource is shown as **Location**.

8. Save the locale-specific version and deploy the form into the runtime environment.

Language-specific and Country-specific Properties Files

The language specific properties file is a copy of the base properties file. This file is renamed using the naming conventions for languages and regions.

Each localized language is represented by a two-letter code, in the format ll, where ll is a lowercase, two-letter ISO 639 language code. For a list of language codes, visit the following web site:

`http://www.loc.gov/standards/iso639-2/langhome.html`

Each country is represented by a two-letter code, in the format CC, where CC is an uppercase, two-letter ISO 3166 country code. For a list of country codes, visit the following web site:

`http://www.iso.org/iso/english_country_names_and_code_elements`

The form name, language code, and optional country code are separated by underscores. The table Renaming Locale-specific Properties Files shows examples of locale-specific `properties` files for a form named **DemoForm**.)

Renaming Locale-specific Properties Files

Filename	Locale description
DemoForm.properties	Original filename. This is the base properties file.
DemoForm_fr.properties	Contains localized strings for the French version of the form. Use this format (without specifying a region) when there is only a single version of the form for this language.
DemoForm_fr_FR.properties	Contains localized strings for the French version of the form used in France.
DemoForm_fr_CA.properties	Contains localized strings for the French version of the form used in Canada.
DemoForm_ja.properties	Contains localized strings for the Japanese version of the form.

As shown in the table Renaming Locale-specific Properties Files, if your form is called DemoForm, the automatically generated base properties file will be called `DemoForm.properties`. This is the file that will contain the strings typed on the form's property sheets.

To create a French version of this form, copy the `DemoForm.properties` file and rename the copy `DemoForm_fr.properties`. This is a language specific variant of the properties file which contains the translation for the French language.

You can also create country specific versions of `DemoForm_fr.properties` file for France and French-speaking Canada. The country specific variant of the properties file contains only those keys for which the translation varies locally in each country.

While creating country specific properties file such as `DemoForm_fr_FR.properties` and `DemoForm_fr_CA.properties`, it is better to create the `DemoForm_fr_FR.properties` and do all the translations. Then copy the latter to `DemoForm_fr_CA.properties` and make the additional changes.

Finally, in both `DemoForm_fr_FR.properties` and `DemoForm_fr_CA.properties` delete all the entries whose keys and values are identical to those in `DemoForm_fr.properties`.

The hierarchy in which the keys are resolved is as follows:
- The keys are first resolved in country specific versions of the properties file such as `DemoForm_fr_FR.properties` and `DemoForm_fr_CA.properties`.
- The keys not provided in the country specific versions are resolved in the language specific version of the properties file such as `DemoForm_fr.properties`.
- The keys not provided in the language specific version are resolved in the base properties file such as `DemoForm.properties`.

If you want to make changes to the labels or messages in the base properties file of your form, and you want corresponding changes to appear in the language specific versions of the properties file, you must make the latter changes manually by editing the strings in the language-specific version of the properties files. An alternative way of doing these changes is as follows:

1. You can select both the base properties and your language specific properties file in Project Explorer and use **Context Menu** > **Compare With** > **Each Other** to open them side-by-side in the Property Compare editor.

2. Use the **Copy All Non-Conflicting Changes** or **Copy Current Change** (From ... To ...) actions to add new keys and delete old keys from your localized version. For new keys and those with updated values you can provide a new translation.

    If the property keys are very similar, the Property Compare editor sometimes misidentifies change types. It is up to you to inspect each change and decide whether the default merge action proposed by the editor is appropriate. If not, you can manually add, delete or amend the localized keys and values instead of using the **Copy Current Change** (From ... To ...) action.

Locale-specific Version of a Form at Runtime

When localized versions of a form exist along with the base version in the runtime environment, the runtime will choose the locale-specific version that corresponds to the locale that is set on the user's system. If no version is present on the runtime server for that locale, the base version will be used.

You can use the `Form.setLocale(String)` and `Form.getLocale()` methods to change the locale settings of the form.

Defining Localization Properties Outside the Form

In addition to creating localized versions of the base properties file of a form, TIBCO Forms supports the creation and localization of additional properties files whose scope is not limited to a given form.

These properties files can be referenced by a form and, in fact, shared by any number of different forms within the same or other projects.

Procedure

1. Create a new resource file, with the extension `.properties`, within the folder **/<project>/Presentation Resources** in the Project Explorer. (This is unlike the base properties file, which is also contained in the **Presentation Resources** folder, but is within a sub-folder for resources specific to the form, a sub-folder named with the name of the form.)

2. Edit the properties file by adding key-value pairs in the format `<key> = <value>`, each on a separate line. For example:
   ```
   mykey1 = My Key One
   mykey2 = My Key Two
   ```

   The format is that of a standard Java resources file, identical to the generated base properties file found in the form folder.

3. Copy the new resource file and save it with the same name but with an underscore and the locale code added before the file extension. For instance, if you wish to create a French version of a properties file named `myResources.properties`, save the first file as `myResources_fr.properties`.

4. In the key-value pairs of the localized version of the properties file, translate or edit the values as desired, while leaving the keys unchanged.

5. The localized version is now available, and can be used as shown in the example that follows.

Example Using a Localization Properties File Defined Outside the Form

A localized properties file can be used within a form. In the example, a button is created that changes the label for a text field. The value for the label is localized using properties files external to the form's own properties files.

Procedure

1. Add a text field and a button to a form.

2. In the **Properties** view for the form, go to the **Resources** tab. Click the plus sign to add a resource, locate the new properties file you created in the **Presentation Resources** folder, and add it as a resource for the form.

   > You will add the new properties file as a form resource using the base name. The various localized versions, with the locale code appended to the file name (preceding the `.properties` extension) will be inferred from the base name, based on the user's locale, at runtime.

   The properties file now appears as a resource in the **Resources** tab, identified by a name and path (URI). The Localized button is automatically selected for the properties file, indicating that the run time should search for localized copies to match the user's locale.

3. In the **Properties** view for the text control, give the control a name in the **Name** field on the **General** tab, for instance **localizedText**.

4. Go to the **Rules** tab in the **Properties** view for the button. Click the button to **Define a new rule** for the button that will be triggered when the button is clicked.

5. Leave the values unchanged in the Rule Details dialog, and click **Next**.

6. Leave the values unchanged in the Rule: Pick Events dialog and click **Next**. This simply means the rule we create will be triggered when the button is clicked, which is the default event for buttons.

7. In the Define Actions dialog, click the plus sign to define a new action.

8. In the Add Action dialog, select the radio button **Create a new action**, and leave the radio button **Script Action** selected. Click **Next** to specify a script that defines the action.

9. Using the content assist pop-ups to ensure correct values, type the following line of script (assuming there is an item in your properties file whose key is **mykey1** and whose value is **My Key One**):

   ```
   control.localizedText.setLabel(resource.
      MyLocalizedResourceFile.mykey1);
   ```

10. Preview the form in the **GWT Preview** tab. Click the button on the form, and the text field's label should say **My Key One**.

11. While still in preview mode, scroll down to the area immediately below the form and change the locale used for the preview from Default Locale to **French - France**.

    > At runtime, the locale of an actual user is set on the user's system or in the user's browser. The locale setting currently is not available for the **GWT Preview**.

12. Click the button on the form again, and the text field's label should now show the localized French text for the button's label.

Business Analysis and Solution Design Modes

There are two capabilities within the Forms Designer in TIBCO Business Studio, Business Analysis and Solution Design. Using the Solution Design capability, you can write scripts for actions and validations, and deploy forms.

If you are in the Business Analysis perspective, you do not have access to:

- The **Deployment Servers** tab in the Project Explorer.

- The script input pane on Actions. Business Analysts can only change the label on Actions.

- The names or Rename Button on Controls, Panes, Actions, and Rules.

- The **Validations** tab on a control's Properties View.

To enable or disable these capabilities, click the "triangle and rule" toolbar button to open the drop-down list that lets you select the desired capability.

Business Analysis and Solution Design Modes
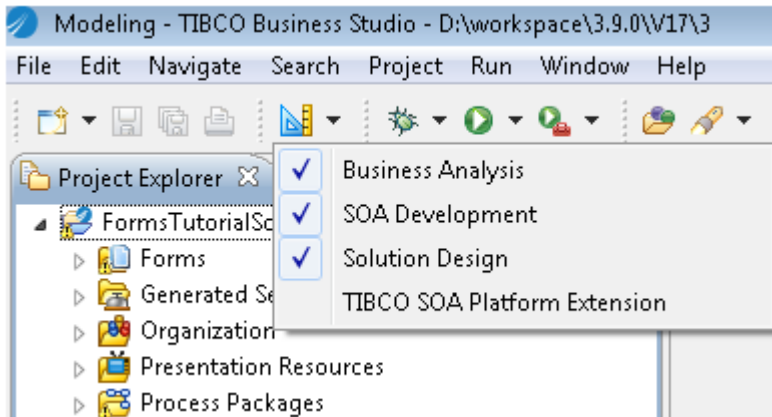
Migration from Previous Versions of TIBCO Business Studio Forms

The form model schema changed from version 1.0 in TIBCO Business Studio 3.9 to version 2.0 in TIBCO Business Studio 4.0. Therefore, forms created in TIBCO Business Studio pre-version 4.0 require migration from form schema version 1.0 to version 2.0.

If you import a form that was created in TIBCO Business Studio pre-version 4.0, it appears with a red X problem marker decoration. If you select the form, the **Problems** view displays the message:
`This resource has an old format and requires migration.`

1. In the Problems view, right-click the marker for the form you want to migrate.

2. Select **Quick Fix** from the context menu.

3. In the Quick Fix popup, select the form(s) you want to migrate, then click **Finish**.

After the migration is finished, the Problem marker decorations are removed from the migrated forms in the Project Explorer view.

The Problems view **Configure Contents... action** allows you to specify what content to display in the view. For each active Configuration, you can filter the view contents by restricting the **Scope**, **Description**, **Severity,** and marker **Types** displayed.

These content restrictions also apply within the Quick Fix popup, so if your intention is to 'quick fix' all instances of a given problem in a given project or the entire workspace, you should ensure that the Problems view contents are configured to include the required resources and marker types. For example, to migrate all forms in the workspace, you would need to have Scope = On any element.

The changes within the migrated forms are:

- **Mapping In** and **Mapping Out** expressions are replaced with *bindings* where possible.

- If a **Mapping In** expression did more than just assign the value of a parameter, that **Mapping In** expression is replaced with a computation action rule triggered by the **Form open** event.

- If a **Mapping Out** expression did more than just assign the value of a control, that **Mapping Out** expression is replaced with a computation action rule triggered on **Form submit** event.

- Event handlers on controls or the form are migrated to rules triggered on the specific control or the form.

- Actions are migrated to script actions.

- If the special file `<project>/<form folder>/META-INF/form_ext.js` is detected during migration, it is added as a JavaScript resource.

- Validations such as during form submit no longer execute validations for controls that are invisible, or are inside panes that are invisible.

- This release includes additional design-time checks. You may see problem markers appear in migrated forms that were not seen in earlier versions of TIBCO Business Studio.

Advanced Tasks

You can improve usability, and enhance the appearance of your forms by performing a few advanced tasks using TIBCO Business Studio Forms.

Using CSS to Customize the Rendering of a Form Control

TIBCO Business Studio Forms supports the use of Cascading Style Sheets (CSS) for customizing how form controls are rendered. You can use CSS with Business Studio Forms to apply styling to a form control.

Explanation

This task covers the case where you want to apply special styling to a specific control in a form.

In order to design the rendering of a control, it is useful to know how the control is rendered in the browser. TIBCO Forms makes use of CSS classes attached to the HTML DOM nodes in order to control rendering. Generally, it is not necessary to know which actual HTML elements are being used in the rendering, and as a practice you should try to use only the CSS classes in devising CSS selectors in your stylesheets, as this approach is the most portable across different target platforms.

Shown here is a representation of the CSS classes that are used to render a control, and their relationship to one another within the nested DOM:

```
-component, customclass
    -label
    -container
        -control
        -hint
```

See Reference for a detailed description of the CSS classes used in rendering forms.

The *customclass* is the name of a CSS class specified in the design time model.

Procedure

1. Create a form that contains one or more controls.

2. Link the form to a custom CSS stylesheet.

> **To create a CSS file in your project**
>
> In the Project Explorer, right-click the **Presentation Resources** folder for your project and click **New** > **File** . The New File dialog opens, where you indicate the parent folder where the CSS file for this form will be contained, and the file name. If there is already a **css** folder within your **Presentation Resources** folder, you can choose that one or, if not, create a folder with that name. But whether you use a subfolder, and if so, what it is named is unimportant. What is important is that the CSS file be placed in or under the Presentation Resources folder and that its filename ends with the extension `.css`. When you click **Finish**, the CSS file is created and opened in the editor.

> **To link a form to a CSS stylesheet**
>
> Be sure the CSS file is already present in the **Presentation Resources** folder. Then, in the **Properties** view for the form, click the **Resources** tab. Click the plus sign (+) to add a resource. The Pick Resource dialog opens, displaying a list of the resources currently residing in the **Presentation Resources** folder, including CSS files, JavaScript, and image files, if any. Select the desired CSS file and click **OK**. Your CSS file has now been added as a resource to your form. The definitions it contains will be used to render the form in HTML.

3. With the form open and visible in the editor, click one of the controls on the form to open the **Properties** view for the control.

4. Enter a name in the **Style Class Name(s)** box on the **General** tab of the **Properties** view for the control.

5. Change the label font properties for this control. For example, add the following lines in the linked CSS stylesheet:

```
.highlight .label,
{
    color: #FF0000;
    font-family: Helvetica, sans-serif;
    font-size: 12px;
    font-weight: bold;
}
```

The CSS selector used here is `.highlight .label`. This is used for clients that use GWT, which is the rendering used in AMX BPM Openspace and Workspace.

6. Put a border around the highlighted control and change the background color. For example, add the following lines to the linked CSS stylesheet:

```
.highlight,
{
    border-style:solid;
    border-width: thin;
    background-color: #DDFFDD;
}
```

Using Editable List Controls

You can bind editable list controls to data parameters of the primitive array data type.

If you have data parameters of the primitive array data type, you can bind the editable list controls to them. You can create action scripts for adding items or for deleting items from the list control. You can also add scripts for validating the values provided in the list control.

Procedure

1. Add new data parameters **strArray**, **intArray**, and **decArray** of the respective types **Text**, **Integer**, and **Decimal**. All of these should be of **array** type.

2. Add three **Text** controls with labels **Text List**, **Integer List**, and **Decimal List** in to the form. Set the names of these controls to **textList**, **integerList**, and **decimalList**.

   For each of these controls:

   Go to the **Properties** tab and select the **Edit as List** check box.

   Go to the **General** tab and add a new binding for the **Value** that points to the value of the respective data parameter array.

3. In the form preview, you will see the three editable list controls.

4. Add a new button **Add Item** to the form.

   Add a new rule for this button and associate following action script for the **Select** event of this button. This script adds the last item into the list.
   ```
   var list = control.textList.getValue();
   list.push("New Value");
   control.textList.setValue(list);
   ```

5. Add a new button **Delete Item** to the form.

Add a new rule for this button and associate the following action script for the **Select** event of this button. This script deletes the last item from the list.

```
var list = control.textList.getValue();
list.pop();
control.textList.setValue(list);
```

6. For the text control named **Text List**, add the following validation script for the **On Value Change** event. This validation is successful when the item added in the list control starts with **Text**. Otherwise, a problem marker appears near the list control.

```
var result = true;
var arr = this.getValue();
if (arr instanceof Array) {
    var length = arr.length;
    for (var i=0; (i<length) && result; i++) {
        if (arr[i].indexOf("Text")==-1) {
            result = false;
            break;
        }
    }
}
result;
```

Also add an error message to be displayed in case the validation fails:

```
Provide input that starts with Text.
```

Changing a Control's Background Color Based on its Value

You can customize the background color of a control using a computation action and CSS classes.

Explanation

This topic covers the case where you want to apply a background color to a given control in a form based on the control's value.

In order to implement this task, you will need to know:

- How to specify a custom CSS document and refer it in the form.

- How to add a computation action that is targeted to a property of the control

Procedure

1. Create a form with one or more controls.

2. Add following classes to the custom CSS document and refer to the document in the form

```
.normalbg,
{
    background-color: #808080;
}
.warningbg,
{
    background-color: #00FF00;
}
.problembg,
{
    background-color: #FF0000;
}
```

3. Add a computation action for the **Style Class Name(s)** property in the **General** Properties view for the form.

Provide following JavaScript code for this action and select the **update** event of this control.

```
var value = parseInt(control.textinput1.getValue());
var bgclass = "normalbg";
if ( value <= 100) {
    "normalbg";
} else if ( value > 100 && value <= 500 ) {
```

```
        "warningbg";
    } else if ( value > 500 && value <= 1000 ) {
        "problembg";
    }
```

4. Preview the form.

   Provide an integer value between 0 - 100 and the background color for the control is set to gray.

   Provide an integer value between 101 - 500 and the background color for the control is set to green.

   Provide an integer value between 501 - 1000 and the background color for the control is set to red.

Performance Improvements

You can improve the performance of forms in TIBCO Business Studio in many ways.

Static Rendering

There are certain cases where the information displayed within a pane is read-only, and the user does not need to edit the values in the pane. In such scenarios, you may gain a performance boost in the load time of the form by marking the pane to use static rendering.

How does Static Rendering Improve Performance?

When a pane is marked to use static rendering, the following optimizations are applied:

- **Faster Rendering**: Form uses an optimized rendering of the controls and markup within the pane which helps the form to render faster.

- **Reduced Load Time**: For a pane having multiple child controls and child panes, individual objects are not instantiated for each child. This reduces the load time considerably. The drawback is that it is not possible to reference those objects using JavaScript in form actions.

Although the static rendering feature helps to enhance the performance of forms it imposes constraints on model validations. The runtime functionality of static panes is also restricted. Refer to Runtime Functionality for details.

When to Use Static Rendering

The use of static rendering may not make a big difference in simple and small panes. The difference in load time is more pronounced as the panes get larger in terms of child controls and child panes.

Using static rendering can be useful in the following scenarios:

- Panes that need to display a large amount of non-editable information.

- Non-editable grid panes, such as those used in a master-detail implementation. It is possible to select individual rows, and the data within the pane are refreshed if the underlying records are modified. However, the static grid pane renders faster than the corresponding editable grid pane.

Configuration of Static Rendering

Panes support the static rendering functionality. You can configure this feature using the options available on the **Properties** tab in the **Properties** view of a pane.

- **Static Rendering**: Check box used to mark a pane to use static rendering. If selected, the pane is rendered as static pane. This property can be set only at design-time. It is not possible to convert a pane to static at runtime.

- **Text Only**: Check box used to mark a static pane to use text-only rendering. If selected, the pane is rendered as plain text, with no control widgets. This check box is enabled only if the **Static Rendering** check box is selected .

Constraints on Model Validations

Panes with the **Static Rendering** property set to `true` have a few constraints on model validation.

The constraints are as follows:

- Static panes are only supported for the GWT desktop runtime.

- Static panes are only supported for grid, vertical, and horizontal panes. Any pane marked as static can contain only these types of panes.

- Controls and panes within static panes cannot be referenced using JavaScript. These controls and panes do not show up in content assist, and any references to these components in JavaScript or computation actions display an error-level problem marker.

- Panes and controls, except button controls, contained within a static pane do not raise events, and thus cannot be used to trigger rules. Events for components within a static pane do not show up as choices for rules.

- Controls and panes within static panes do not support computation actions.

- Controls and panes within static panes do not support validations.

- Controls and panes within static panes do not support bindings to properties. However, binding to the following features are supported:

  - Values

  - Choice values

  - Labels of optionlist

  - Radiogroup

  - Hyperlink

  - Linktext

  - Image URL

- Panes contained within static panes are also considered static panes.

- Tab order is ignored on controls within static panes.

- Values on controls and panes in static panes support absolute bindings and absolute ancestor pane value bindings to data fields and parameters. Bindings to other controls in the form are flagged with an error-level problem marker.

- Static panes cannot contain tabbed, grid, record, or message panes.

- Static panes cannot contain embedded forms.

- The **Static Rendering** property setting is ignored by the Mobile runtime. A warning-level problem marker is shown if a pane has the **Static Rendering** property set to `true` and any of the presentation channels uses Mobile rendering.

  > Top-level static panes can be referenced in form action scripts, computation action destinations, and bindings. But, nested panes and nested controls cannot be referenced.

Restrictions on Runtime Functionality

For static panes, contents of the pane are rendered in simple HTML using streamlined JavaScript generated at design-time.

- Validation markers are not displayed on controls in static panes.

  - Initial data are assumed to be valid.

  - For master-detail configurations, the grid pane can be updated using a non-static detail pane, but validation markers are only shown in the detail pane.

  - Data can be changed using the data API.

    > ⚠️ The values updated using the data API are not validated if they are shown only within a static pane.

- 'Required value' indicators are not displayed on controls in static panes.

⚠ Static panes should only be used in cases where you are assured that all required values have already been filled out, or when the user has an alternate method of specifying data, such as master-detail configurations. An example would be a step in a process where a user confirms previously specified data before proceeding.

- Controls in static panes are completely static. It is possible to set a Style Class Name on a static pane and the child components, but the value is fixed at design-time.

Pane Value Update

When the value of a static pane is updated using either script, binding or computation action, the content of the pane is regenerated using the same JavaScript initially used to render the pane.

⚠ A control within a static pane will not be refreshed when the underlying data value is updated if the control is directly bound to either of the following:
- A primitive parameter or data field.
- A primitive attribute of a data field.

Static Grid Panes

Static grid panes support the following functionality:

- Row selection
- Pagination
- Adding records
- Deleting records
- Sorting

The following functionality is not supported in static grid panes:

- Editing
- Validations on controls
- Computation actions on controls

A static grid pane is rendered as a compact non-editable grid pane, with the values represented as plain text.

It is possible to use a static grid pane as a part of a master-detail configuration. The non-static detail pane can be bound to the selection of the grid pane as is currently done. When a value is changed in the detail pane, the corresponding row in the static grid pane is re-rendered using the original generated JavaScript.

Tabbed Panes

Although tabbed panes cannot be marked as static, child panes that are vertical, horizontal, or grid panes can be marked as static.

Localization

Static panes support localization and will be regenerated if the form locale is updated.

Renderings for Specific Controls

Most controls in a static pane are rendered in the same fashion as in a normal pane, but are rendered in a read-only fashion.

If the **Text Only** property is set to `true`, then the value of each control is rendered as plain text. The values are formatted appropriately according to the type (as listed in the table Rendering of Specific Controls). The control widgets are not rendered. Although, the rendering of images, hyperlinks, buttons, and pass-through controls is the same as in a static pane.

The table Rendering of Specific Controls lists how specific controls in a static pane are rendered:

Rendering of Specific Controls

Control	Rendering in Static Panes
Text	Rendered as a read-only text input.
Text-Secret	Rendered as a read-only secret text input (values are obscured).
Text-Numeric	Rendered as a read-only text input. Numbers are formatted according to the format set on the control.
Textarea	Rendered as a read-only text area. The content of the text area is scrollable.
Checkbox	Rendered as a read-only check box.
Date	Rendered as a read-only input. Value is formatted using the date format.
Time	Rendered as a read-only input. Value is formatted using the time format.
DateTime	Rendered as a read-only input. Value is formatted using the datetime format.
Duration	Rendered as a read-only input; formatted as is done for the read-only view in grid panes. For example: 3 hours, 15 minutes.
Hyperlink	Rendered as a normal, active hyperlink.
Image	Rendered within an img element.
Label	Rendered as plain text.
Optionlist	The label for the selected value is displayed in a read-only input element.
Multi-select Optionlist	A read-only version of the multi-select optionlist is displayed, with the selected values highlighted.
Pass-through	Static pass-through content is inserted as normal.
Radiogroup	Rendered as a read-only radiogroup, showing the selected value.
Button	Rendered normally. The button is active and can trigger rules defined in the form model.
List controls	Values rendered in a string, in a read-only input, using the localized list item-separator.

Deferred Rendering and Deferred Initialization

The initial load time for complex forms can hinder the user experience. There can be a delay if the user interface is initially hidden within the tabs of a tabbed pane. In such scenarios, using deferred rendering or deferred initialization of panes can help to reduce initial load-time.

By using these features, the rendering of panes on a page is deferred until after the basic framework of the form is loaded and is operational.

How do Deferred Rendering and Deferred Initialization Improve Performance?

When a pane is marked to use deferred rendering or deferred initialization, the following optimizations are applied:

- **Deferred Rendering**: The rendering of the pane is deferred till the pane is made visible by the user. The panes that are visible at initial load-time are rendered when:

    – The form is completely initialized.

    – Form open event has fired.

    – All the form open rules have been executed.

    In tabbed panes, the rendering of each tab is deferred until the user clicks on the tab to view the contents.

- **Deferred Initialization**: The deferred initialization feature can only be used for panes that are marked to use deferred rendering. The children of the pane marked to use deferred initialization are not initialized until the pane needs to be rendered. This means that the pane object itself is always instantiated and available, but any nested children are not initialized.

> Deferred initialization imposes restrictions on the types of references that can be made to the child controls of the pane. Refer to Deferred Rendering and Deferred Initialization Constraints for details.

Configuration of Deferred Rendering and Deferred Initialization

Panes support the deferred rendering and deferred initialization functionality. You can configure these features on the **Properties** view of a pane. Both the options are available on the **Properties** tab.
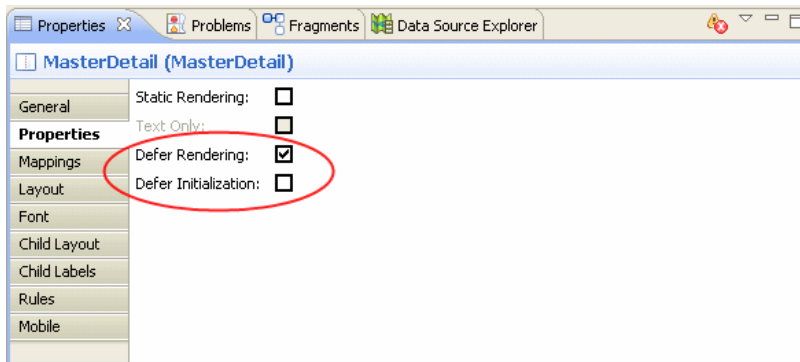
- **Defer Rendering**: Check box used to mark a pane to use deferred rendering. If selected, the user interface for the pane is not rendered until the pane is made visible. This property can be set only at design-time and it cannot be updated using bindings or using the API.

- **Defer Initialization**: Check box used to mark a pane to use deferred initialization. This check box is enabled only if the **Defer Rendering** check box is selected. If selected, the children of the pane are not initialized until the pane needs to be rendered.

Constraints on Model Validations

Panes with the **Deferred Rendering** property set to `true` have a few constraints on model validation.

- Deferred rendering of a pane is supported for the GWT runtime.

- The **Deferred Rendering** property setting is ignored by the Mobile runtime. A warning-level problem marker is shown if a pane has the **Deferred Rendering** property set to `true` and any of the presentation channels uses Mobile rendering.

- When a pane is marked for deferred initialization, all references to child or nested controls of that pane are flagged with an error-level problem marker. This includes references in script or computation actions. The following quick fixes are available:

  – Remove deferred initialization.

  – Use Defer Rendering only.

- Panes marked for deferred initialization cannot contain embedded forms, either directly or in any of the nested panes. This is indicated by an error-level problem marker. The following quick fixes are available:

  – Remove deferred initialization.

  – Use Defer Rendering only.

Modal dialog panes cannot be marked for deferred rendering or deferred initialization. However, a child pane of a modal dialog pane can be marked for deferred rendering or for deferred initialization.

Restrictions on Runtime Functionality

For deferred panes, you may observe a few restrictions on runtime functionality.

Handling Bindings to Deferred Panes and Child Controls

- If the **Defer Initialization** check box is cleared, then bindings, script references, and computation action references to the pane and its children are not affected.

- If the **Defer Initialization** check box is selected, then any references to child or nested controls using scripts or computation actions are flagged with an error-level problem marker. You can make use of events tied to the pane and its children in rule definitions. Binding to panes are always active and working, but the bindings to child and nested controls are inactive until the pane and child controls have been fully initialized.

- If the **Defer Rendering** check box is selected, and the **Defer Initialization** check box is cleared, bindings to panes and controls are active even if the pane is not currently rendered. The internal model of the pane or child controls can be updated using scripts, bindings, or computation actions. The effects of such updates are visible after the pane is rendered.

Handling Validations in Deferred Panes

An un-rendered pane is treated the same as an invisible pane with respect to the suppression of validation checking.

Loading Deferred Panes

Panes marked to use deferred rendering display a spinning wheel to indicate that the content is being initialized. This loading indicator is visible only if there is a noticeable delay in rendering the pane.

Custom Controls

TIBCO Business Studio supports integration of third-party custom controls. Users can provide configuration information about any third party widgets, and can expose those controls in the palette. You can work with these extended controls in the same fashion as you do with the set of built-in controls.

Definition of Custom Controls

There are two key items that a developer needs to provide for the definition of a custom control - a Control Wrapper, and the definition of the custom control in the component library file.

- A ControlWrapper is a JavaScript class that either implements the runtime functionality of the control, or wraps a third-party library. The figure below provides a look at how the ControlWrapper exposes the implementation of a third-party library as a Custom Control within Forms.

  > It is also possible to provide the entire implementation of a custom control within a ControlWrapper with no reliance on a third-party library. However, that is not the typical case.

  Custom Control Architecture
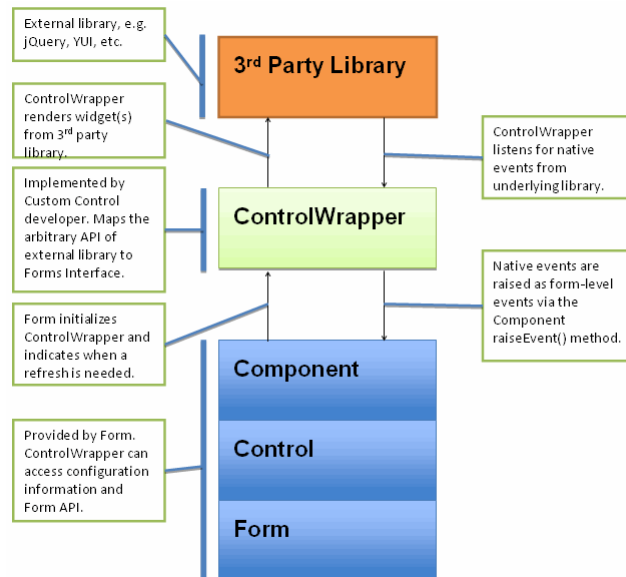
- The custom control definition must be specified in a component library file. The component library file provides information on how to display and configure instances of the custom control in the Form Designer. The information will also be used at runtime in order to determine the capabilities of the control.

  The figure below provides a description of the various design-time and runtime artifacts that go into a Components Library project.

Component Library Project

The following annotations appear beside the Project Explorer diagram:

- Control Type values and properties can reference dataTypes defined in a BOM
- Icons used in the Form Designer palette and canvas are located in the Components special folder.
- The Components Library model contains definitions of custom ControlTypes
- All resources needed at runtime need to be in the Presentation Resources special folder. It is a good practice to group these within a single folder to avoid conflicts with the resources of multiple Component Libraries.
- Custom resources can include CSS files and localized resources.
- Any required 3rd-party libraries need to be located in the Presentation Resources folder.
- The ControlWrapper implementations also need to be in the Presentation Resources folder, and referenced from the Component Library

Working with the Component Library File

A special folder of type **Components** is used to store component library files. A library file defines a set of custom controls which are available in the Forms Designer palette.

The option to create the **Components** special folder is presented at the time of new project creation.

Procedure

1. Click **File** > **New** > **BPM Developer Project** . The New BPM Developer Project dialog opens.

2. Specify the **Project name** and select the **BPM** check box as the **Destination Environments**. Click **Next**.

3. The **Form Component Library** option is provided on the **Asset Type Selection** page. Specify the asset types.

   > A Business Object Model asset is only required if you wish to add or use model types that will be used in the component library.

4. The **Asset Type Selection** page provides the following two options for creating a component library project:

Result

- Click **Finish**: creates a new project with a **Components** special folder. The `<library>.library` file is created in the **Components** special folder.

- Click **Next**: displays a wizard page that guides you to create a new component library project.

    1. Specify **Folder** and **Filename** on the **Business Object Model** page and click **Next**.

    2. Specify **Folder** and **Library filename** on the **Component Library** page and click **Next.**

    3. Specify **Folder** details on the **Set Special Folders** page and click **Finish**. This is an optional step. You can also click **Finish** in the preceding step. Creates a new project with a **Components** special folder. The `.library` file is created based on the details provided in the wizard.

       You can designate a normal folder as a **Components** special folder as well, using a similar 'Special Folders > Use as Components Folder' technique as with other special folder types.

    The contents of the **Components** special folder are:

    - `.library` file: the `.library` file contains the configuration information for a set of custom controls. For example: `MyComponents.library`.

    - `icons` folder: the `icons` folder contains sample design-time icons for the custom controls.

1. Right-click the `<library>.library` file, and select **Open**. The library file is opened in the **Component Library Editor** for editing.

    An overview of the various parts of the Component Library Model is provided in the figure Component Library Model.

Component Library Model

The editor supports editing of the `.library` file, and provides an easy way to specify the configuration details for each custom control definition.

2. Select the <Librarynode to view and edit the configuration details for the library element in the Properties view. Refer to Library section for a detailed description.

> The .library file displays a problem marker for defining the Constructor Class property.

The library element can have the following child elements:

- Palette Drawer: a Library has a single Palette Drawer element. Refer to Palette Drawer section for the details.

- Event Type: a Library can have multiple Event Type elements. Refer to Event Type section for the details.

- External Resource: a Library can have multiple External Resource elements. Refer to External Resource section for the details.

- Control Type: a Library can have multiple Control Type elements. Refer to Control Type section for the details.

> 📋 By default, the following elements are added to the Library root element:
> - Palette Drawer
> - Control Type
>
> The other supported elements can be added according to your requirements.

3. To add Event Type and External Resource elements at the Library level:

- Event Type: select the Library element, right-click, and select **New Child** > **Event Type**. A new Event Type element is added.

- External Resource: select the Library element, right-click, and select **New Child** > **External Resource**.

  A new External Resource element is added. An External Resource defined at the Library level applies to all Control Types defined in the Library. It gets loaded into the page if the form uses at least one control type from the library.

4. Select each library element's child elements to view and edit the configuration details in the Properties view. In the case of properties which refer to elements in the workspace, the cell editor consists of a hyperlink and a '...' button to activate the associated property value editor dialog. The figure Library Editor Properties View provides more details.

Library Editor Properties View
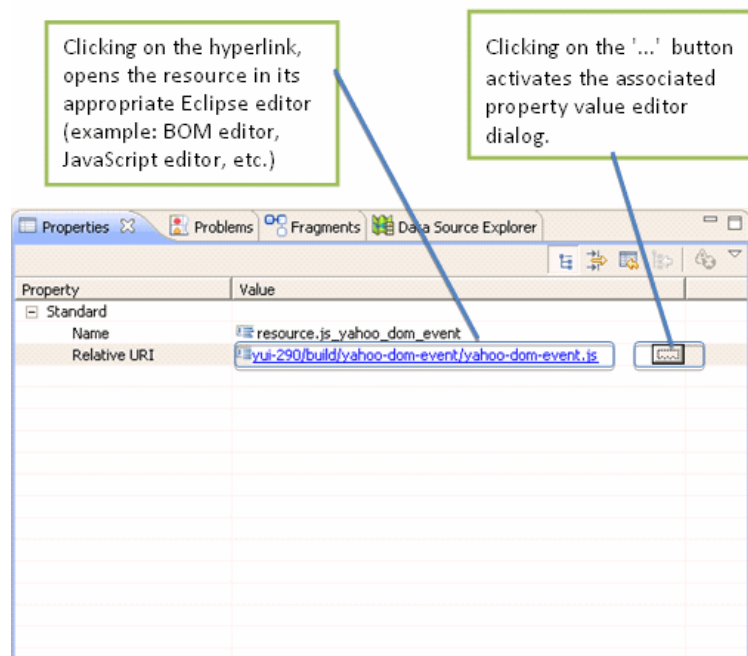
5. The Control Type element can have the following child elements:

- Capabilities: a Control Type has a single Capabilities element. Refer to Capabilities section for details.

- External Resource: a Control Type can have multiple External Resource elements. The properties are same as for an External Resource element at the Library level.

- Property: a Control Type can have multiple Property elements. Refer to Property section for details.

> By default, only the Capabilities element is added to the Control Type node. The other supported elements can be added as per your requirements.

6. To add External Resource and Property elements at the Control Type level:

    - External Resource: select the Control Type element, right-click and select **New Child** > **External Resource**. A new External Resource element is added. An External Resource defined at the Control Type level is guaranteed to be loaded into the page only when a form uses this control type from the library.

    - Property: select the Control Type element, right-click and select **New Child** > **Property**. A new Property element is added.

7. Select each Control Type's child elements to view and edit the configuration details in the Properties view.

Working with the ControlWrapper

You can add a reference to a ControlWrapper class.

Procedure

1. Create a folder in the Presentation Resources folder and place the `.js` file with the JavaScript wrapper implementation in this folder.

2. Select the External Resource element (either at the Control Type level or Library level) to view the configuration options in the Properties view.

3. Click the picker provided for the **Relative URI** property.

    The Pick Resource dialog lists the JavaScript files available in the Presentation Resources folder.

    > You have to select the **Relative URI** property sheet entry in order to activate the cell editor. Once activated, the '...' button opens the resource picker dialog.

4. Select the ControlWrapper class implementation file and click **OK**.

    Refer to Control Wrapper Implementation section for details.

Usage of Custom Controls

Once the custom component definition is complete, the project with the component library file has to be added as a project reference in a Forms project. The icons for the custom components are displayed in the Form Designer palette.

> When adding a project reference to a library project, forms that are open in the referring project will not immediately reflect the new palette drawers available from that project. You will need to close and re-open those forms in order to see the new palette drawers.

The Form Designer supports the following functionality when working with custom controls:

- You can create a new control instance from the palette by clicking the custom control type, moving the mouse to the desired position on the form canvas, and clicking again.

- Controls of other types can be refactored to custom control types by changing the control type field on the **General** Properties tab for the control.

- The Form Designer Properties tab presents a table for editing the extended properties defined for the custom control.

- The preview makes use of the custom controls.

- Custom control libraries are made available by adding a reference to the library project.

- All built-in control functionality is available, unless specifically prevented by the custom control type definition. For example: validation, labels, hints, visibility, and data binding.

- Deployment of custom controls is handled the same as for other BPM projects.

Runtime Life Cycle of Custom Controls

The ControlWrapper implementation is subject to a very specific life cycle, which is described in the figure ControlWrapper Life Cycle.

ControlWrapper Life Cycle
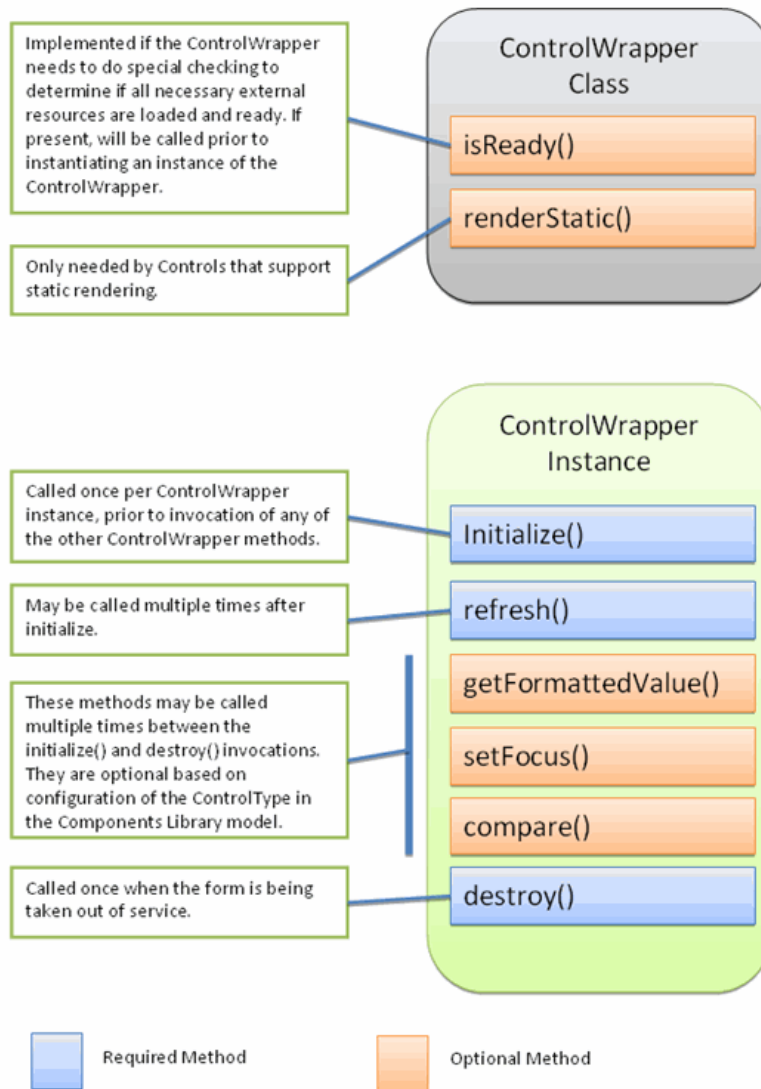
Preparation

The forms runtime repeatedly calls the ControlWrapper's isReady() method until it returns `true`.

Initialization

The forms runtime calls the constructor function to create an instance of the ControlWrapper and then calls the initialize() method on this instance.

Refresh

Whenever you update the configuration or value of the control, the form invokes the `ControlWrapper.`refresh() method to give the ControlWrapper a chance to update the rendering of the control.

Destruction

When the form is being taken out of service, it invokes the destroy() method on the ControlWrapper.

If a control is within a static pane, the form does not create an instance of the ControlWrapper. Instead, it invokes the renderStatic() for the ControlWrapper to get the markup used in the static mode.

Runtime Life Cycle of Custom Control Used within Grid Pane

Using a custom control within a grid pane uses a specialized life cycle. It depends on the supported Render Modes of the Control Type and the **Always Render** setting on the Control.

If a grid pane is not in the Always Render mode, only one instance of the ControlWrapper gets created for a column. Each cell in that column shares that single instance. When the runtime invokes methods, such as refresh() and getFormattedValue() on the wrapper, the getControl() method on the component interface returns the specific instance for that cell. This way, the ControlWrapper implementation gets specific configuration details of that cell.

Preparation

The forms runtime repeatedly calls the ControlWrapper's isReady() method until it returns true.

Initialization

When you configure the control in the **Always Render** mode, the form creates an instance of the ControlWrapper for each cell in the grid table. Users can view the rendering that happens at this point.

When you do not configure the control in the **Always Render** mode (default), the form creates a single ControlWrapper instance that is shared between the cells in a grid table column. Users cannot view the rendering that happens at this point. The form does not attach the parent node to the DOM at this point but invokes the ControlWrapper.getFormattedValue() method for each visible cell in the column of the grid table.

Refresh

When you configure the control in the **Always Render** mode, the form calls the refresh() method as soon as the value or other configuration settings are changed.

When you do not configure the control in the **Always Render** mode and focus on a cell to edit the value, the form calls the ControlWrapper.refresh() method to allow the ControlWrapper to update the rendering and reflect the current value being edited. If you change any of the configuration settings after the last refresh(), the wrapper is notified through the updates argument.

Sorting

If a control manages a complex value, the static compare() method on the ControlWrapper class is called each time user sorts the grid on that column.

Destruction

When you configure the control in **Always Render** mode, the form invokes the destroy() method on all instances of the ControlWrapper when the form is being taken out of service or on the specific instance when that row is deleted from the table.

If you do not configure the control in the **Always Render** mode, the form invokes the destroy() method on the shared instance of the ControlWrapper only when the form is being taken out of service.

Component Library Model

For each library definition in the component library model, there is a palette drawer that displays all the controls defined in the library. In addition to the built-in events, you can also define custom events at the library level.

Library

The Library is the root element in the component library definition.

Library Element Properties

Property	Type	Initial Value	Description
Name	String	Library file base name	Library name.
Qualifier	String	Generated based on containing project identifier and the nsPrefix	The qualifier should be unique within the workspace.
XML Namespace Prefix	String	*<libraryName>*. toLowerCase()	Used in form models when referencing Control Types in this model. This is pre-populated based on the original name of the library file and usually does not need to be changed (it can be manually abbreviated to something short and intuitive).
XML Namespace URI	URI	Generated based on the library name	Defines a unique namespace URI reference that is used in referring to form definitions. This is pre-populated based on the original name of the library file and usually does not need to be changed (it can be manually changed to conform to your organization's end-point naming policy).
Drawer	Palette Drawer		Provides basic information about the drawer in which custom controls in this library will be displayed in the Form Designer palette.
Event	Event Type		Defines custom events that are raised by one or more control types defined in this library. The multiplicity of this property is 0..* (that is, the library can contain zero or more event types). Note: An Event Type is added to the model using the context menu (right-click) on the Library element.

Property	Type	Initial Value	Description
Resource	External Resource		The external resources can contain JavaScript, CSS, image files, or localized properties. All external resources defined in the library are loaded when one of the control types defined in it is loaded by the form. The multiplicity of this property is 0..* (that is, the library can contain zero or more external resources). Note: An External Resource is added to the model using the context menu (right-click) on the Library element.
Component	Control Type		Specifies the Control Type schema. All Control Types specified here appear in the Palette Drawer defined for this Library and can be used in forms. The multiplicity of this property is 1..* (that is, the library can contain one or more Control Types). Note: A Control Type is added to the model using the context menu (right-click) on the Library element.

Palette Drawer

There is a single Palette Drawer for each library file. All controls defined in a library file are displayed in this drawer in the Form Designer palette beneath the built-in drawers of **Panes**, **Controls**, and **Action Buttons**.

Palette Drawer Properties

Property	Type	Initial Value	Description
Name	String	`drawer.<libraryName>`	Defines a unique name for the drawer. The name must begin with "drawer." and must be unique within the workspace. This property is pre-populated based on the original name of the library file and usually need not be changed.
Label	String	Generated based on library name.	Label applied in the form designer.
Order	Integer	0	Specifies the order in which the drawer is added to the palette. Higher numbered drawers appear later in the palette. Any drawers that have the same order are arranged alphabetically by label. Custom drawers are always below the built-in drawers.

Event Type

Custom controls can raise any of the built-in events (for example: **Update**, **Enter**, **Exit**, **Select**) or can raise custom events that do not correspond to the semantics of any of the built-in events. Individual

control types can declare the events they support from the union of built-in events and those defined at the library level.

Event Type Properties

Property	Type	Restrictions	Description
Name	String	- Unique within library. - Cannot be set to a same name as any of the built-in event names: **close** **doubleclick** **enter** **exit** **localize** **open** **select** **submit** **update** - Cannot begin with **tibco_**, to ensure no conflict with built-in events specified by TIBCO.	Specifies the name of the custom event. This is used at design time to configure the events that trigger a rule. At runtime, it is used in the ControlWrapper to specify which event is being raised by the custom control.
Label	String		Label used in the Form Designer for the Event Type.

External Resource

An External Resource can be defined either at library level or at control type level.

- Library level: in this case, it applies to all control types defined in the library and these External Resources are loaded when at least one Control Type in this library is used by the form.

- Control Type level: in this case, it applies only to the specific Control Type and is loaded only if this Control Type is used by the form.

All External Resources needed to load a Control Type need to be defined in one or other of these two places. This includes the JavaScript file that contains the implementation of the ControlWrapper for the Control Type. An External Resource can also contain CSS or localized properties.

External Resource Properties

Property	Type	Initial Value	Description
Name	String	`resource.resource1.` The number increases for each new resource added to the library file.	Short identifier of the resource. Resources should be renamed to something meaningful. The name must begin with "`resource.`" and must be unique within the library file. For properties files, this name is used from script in order to reference the resource bundle. (For example `resource.<name>.<key>`)
Relative URI	String		A URI relative to the Presentation Resources folder. For example, "css/myControl.css". A picker is also available to select the external resource. **Important:** you cannot select an external resource from a referenced project. The external resource must be available locally.

Control Type

The Control Type defines the custom control. A library file can contain one or many Control Types. All Control Types specified here appear in the Palette Drawer defined for the library.

The detailed description of the properties is provided in the table Control Type Properties:

Control Type Properties

Property	Type	Restrictions / Initial Value	Description
Canvas Icon	String	Must be a valid relative URI that resolves to an image file in the **Components** special folder.	(Required) Provides the special folder relative URI of the icon that is used when rendering the component in the Form Designer canvas. The icon has to be placed within the **Components** special folder and it can be an image of type .png, .gif, or .bmp. This icon is used only at design-time. When a library is first created, a set of initial icons is provided in the icons folder. These icons can be used as placeholders for the three icons needed on a Control Type until a more specific set of icons can be provided.
Constructor Class	String	Must be a valid JavaScript expression that yields a constructor function object when evaluated at runtime.	(Required) Refers to the name of the JavaScript constructor that implements the ControlWrapper interface. The JavaScript file that defines this constructor should be specified as an External Resource reference either at the Library level or the Control Type level.

Property	Type	Restrictions / Initial Value	Description
Data Type	Classifier	Must be a BOM Primitive Type, Enumeration, or Class	Defines the type of the value managed by this Control Type. This is a reference to one of the following types: • A built-in BOM Primitive Type • An Enumeration • A Class The Data Type determines what can be bound to the value of controls of this Control Type in the Form model. If the BOM Primitive Type of "Object" is specified for the Data Type, then it allows any complex object to be bound to the value of instances of the Control Type.
Event	Event Type Reference	Can only reference a built-in Event Type, or Event Types specified in this Library file.	Specifies which Event Types can be raised by this Control Type. The runtime ControlWrapper can only raise an event of a given type if it has been declared in the Control Type model. The events specified here will be available in the Form Designer to add as triggers on Rules defined in the Form.
Handles Enter Key	Boolean		This is set to `true` if the underlying widget provides a key handler for the Enter key. The form needs to know this in order to prevent a primary button (for example, 'Submit') from being activated by an Enter keystroke when the custom control has the input focus.
Hint	String		(Required) Short description of the Control Type, which is used as a hower help on the icon in the palette.
Label	String	The library designer should ensure that this is unique within the Library.	(Required) Label used in the Forms Designer palette.

Property	Type	Restrictions / Initial Value	Description
Multi-valued	Boolean		Indicates whether the value managed by this Control Type is multi-valued. If `true`, then the value for the control can only be bound to multi-valued values. When this is `true`, it is up to the implementation of the control to manage multiple values. For a multi-valued control with a simple data type, the runtime value will be set as a JavaScript array. When the control is managing multi-valued structured types, the values will be provided in a list.
Name	String	- Unique within library. The qualified name should be unique in the workspace. - Name must begin with `controls`. The initial value is set to `controls.control 1`	(Required) This is the name of the Control Type that is used when adding a reference from a Form model. The form uses the fully qualified control type name, prefixed by the library qualifier, to avoid name collisions.
Palette Icon 16	String	- Must be a valid relative URI that resolves to an image file in a **Components** special folder. - Must not be the same as Palette Icon 24 - Can be an image of type `.png`, `.gif`, or `.bmp`	(Required) The special folder relative URI of the small (16x16 pixels) icon that is used when rendering the component in the Form Designer palette. The icon has to be placed within the **Components** special folder. This is used only at design-time.

Property	Type	Restrictions / Initial Value	Description
Palette Icon 24	String	• Must be a valid relative URI that resolves to an image file in a **Components** special folder. • Must not be the same as Palette Icon 16 • Can be an image of type `.png`, `.gif`, or `.bmp`	(Required) The special folder relative URI of the large (24x24 pixels) icon that is used when rendering the component in the Form Designer palette when the 'Large Icons' option is active. The icon has to be placed within the **Components** special folder. This is used only at design-time.
Qualified Name			This is a read-only property which provides the <library-qualifier>.<element-name> details.

Property	Type	Restrictions / Initial Value	Description
Render Mode	String	• Render Modes supported for Control Types are: – **normal** – **static** – **view-text** – **view-html** – **grid-edit** • All Control Types must support **normal** mode. • A Control Type can support only one of **view-text** or **view-html**.	Multi-valued enumerated type that defines the render modes supported by the ControlWrapper. The values are: • **normal**: single instance rendering of the control, such as within vertical and horizontal panes. • **static**: Indicates that the control can be rendered within static panes. If supported, then the ControlWrapper must provide the `renderStatic()` method. • **view-text**: If specified, then the ControlWrapper must provide a `getFormattedValue()` method that will return a plain text representation of the value managed by this control. • **view-html**: If specified, then the ControlWrapper must provide a `getFormattedValue()` method that will return an HTML representation (as a string) of the value managed by this control. • **grid-edit**: Indicates that the Control Type provides a rendering specific to edit mode of grid panes. If this mode is not specified, and the Control Type otherwise supports grid panes, then the normal rendering mode will be used in grid panes. The value of `getFormattedText()` is used in grid panes in the view mode. If neither **view-text** or **view-html** is specified, then the **grid-edit** mode will always be used in grid panes, or will fall back to **normal** if **grid-edit** is not specified. The Focus capability must be defined for the **grid-edit** mode.

Property	Type	Restrictions / Initial Value	Description
Supported Parent Pane	Pane Type Reference	Only one of Supported Parent Pane or Unsupported Parent Pane references can be used within a given Control Type.	Specifies a list of pane types that are supported as a direct parent by this Control Type. A control of this Control Type can only be added to panes of types on this list. If neither Supported Parent Pane or Unsupported Parent Pane restrictions are specified, then it is legal to add an instance of this Control Type into any type of pane that will accept it as a child.
			Some pane types restrict the type of children they support. For example, grid panes don't allow panes as children; tabbed panes only allow panes as children; an embedded form pane is only allowed to refer another form in the workspace.
			Note: No Control Types are supported in the Message Pane. You will not be able to place any Control Type in a Message pane even if it is selected as one of the Supported Parent Pane types.
Supported Type Conversion	Control Type Reference	Only one of Supported Type Conversion or Unsupported Type Conversion references can be used within a given Control Type.	Specifies an explicit list of Control Types to which an instance in a form may be refactored. If not specified, an instance of this Control Type may be refactored to any Control Type. For example: a third party date picker may only permit itself to be refactored to one of the built-in date-time control types.
Unsupported Parent Pane	Pane Type Reference	Only one of supported Parent Pane or unsupported Parent Pane references can be used within a given Control Type.	Specifies a list of pane types that this Control Type does not support as a direct parent. If a pane is included in this list, then it is not possible to place a control of this type into an instance of that pane. If neither Supported Parent Pane or Unsupported Parent Pane restrictions are specified, then it is legal to add an instance of this Control Type into any type of pane that will accept it as a child.
Unsupported Type Conversion	Control Type Reference	Only one of Supported Type Conversion or Unsupported Type Conversion references can be used within a given Control Type.	Specifies an explicit list of Control Types to which an instance in a form may not be refactored. If not specified, an instance of this Control Type may be refactored to any Control Type. For example: a third party slider control may forbid itself to be refactored to a tree control in the same component library.

Capabilities

Each capability is specified by an enumerated list comprised of neither, either, or both of the values [`component`, `form`].

- `component` flag: The presence of this flag indicates that the component will provide some level of functionality with regards to that capability, so it should be provided with the necessary information and notified if the information related to that capability is updated.

- `form` flag: The presences of this flag indicates that the component expects the form to carry out its normal handling of the capability, even if the component flag is also specified for the capability.

The table Flagging and Outcomes provides specific detail for each combination of flags for each of the capabilities.

Flagging and Outcomes

Property	Description	Form Flag	Component Flag	Outcome
Disabled	The form will not have enough information to know how to disable a widget within the custom control. If a Control Type is to support the setting of a disabled state, then it will have to handle the update of this property at runtime.	`true`	`true`	This is the typical case. Here, the form applies or removes the "disabled" CSS class at the control level, and requests the ControlWrapper to refresh its rendering of the enablement state. The form notifies the ControlWrapper that the enablement state has changed by calling its `refresh()` method with the `updates` argument containing the feature name "enabled".
		`true`	`false`	The form will apply or remove the "disabled" CSS class at the control level but does not notify the ControlWrapper of enablement changes.
		`false` (default)	`true` (default)	No CSS class is applied at the control level but the form notifies the ControlWrapper of enablement changes.
		`false`	`false`	The control does not handle the disabled state.
Focus	For this capability, the `form` value is always set to `false`	`true`	`true`	N/A
		`true`	`false`	N/A

Property	Description	Form Flag	Component Flag	Outcome
		false (default)	true (default)	ControlWrapper supports a `setFocus()` method to allow script to change the focus to the control programmatically. If the control type implicitly or explicitly supports rendering within a grid pane, then the focus capability should be set to "component". Otherwise keyboard navigation of the grid pane will skip over cells that contain instances of this control type.
		false	false	ControlWrapper does not provide `setFocus()` method.
Hint	This capability controls how the Control Type hint is handled. This is the built-in control hint that is provided by the forms framework.	true	true	The form renders the hint as normal, and requests the ControlWrapper to refresh its custom hint rendering. The form notifies the ControlWrapper that the hint state has changed by calling its `refresh()` method with the `updates` argument containing the feature name "hint".
		true (default)	false (default)	This is the typical case. The form renders the hint as it does for built-in controls but does not notify the ControlWrapper of hint changes.
		false	true	The hint node is not rendered by the form. It is completely up to the ControlWrapper to handle the rendering of the hint.
		false	false	The hint node is not rendered for the Control Type.

Property	Description	Form Flag	Component Flag	Outcome
Invalid	This Capability controls how the rendering of "Invalid" feedback is handled. The forms framework continues to execute validations on controls that provide them.	true	true	The form applies or removes the "invalid" CSS class at the control level, and requests the ControlWrapper to refresh its rendering of the validity state. This may be needed by controls that aim to provide accessibility. For example, by updating the corresponding ARIA attributes on the control widgets. The form notifies the ControlWrapper that the validation state has changed by calling its `refresh()` method with the `updates` argument containing the feature name "validation".
		true (default)	false (default)	The form applies or removes the "invalid" CSS class at the control level, but does not notify the ControlWrapper of validity changes.
		false	true	No CSS class is applied at the control level but the form does notify the ControlWrapper of the change in validity state.
		false	false	The control does not handle the display of a validation error decoration.
Invisible	This Capability handles how the visibility setting of the Control Type is handled.	true	true	The form hides or shows the whole control and also notifies the ControlWrapper that the visibility state has changed by using the `refresh()` method with the `updates` argument containing the feature name "visible".
		true (default)	false (default)	The form takes care of hiding and showing the control when the visibility state has changed but does not notify the ControlWrapper of visibility changes.
		false	true	The form merely notifies the ControlWrapper of changes in visibility.

Property	Description	Form Flag	Component Flag	Outcome
		false	false	The control is always shown. Changes to the visibility of the control are ignored. However, if the containing pane is made invisible, then the control will be made invisible.
Label	This Capability controls how the control label is handled. This is the built-in control label that is provided by the forms framework. When a custom control is rendered in a grid pane, the column label is always provided by the form.	true	true	The form renders the label as normal, and also requests the ControlWrapper to refresh its rendering of the label. The form notifies the ControlWrapper that the label value has changed by calling its `refresh()` method with the `updates` argument containing the feature name "label".
		true (default)	false (default)	The form renders the label as it does for built-in controls but does not notify the ControlWrapper of label changes.
		false	true	The label is not rendered by the form and the form notifies the ControlWrapper of label changes. It is completely up to the ControlWrapper to handle the rendering of the label. In vertical panes, this setting will result in control being rendered completely to the left, aligned with the labels of other controls that rely on the form to render the label.
		false	false	The label is not rendered for the control.
Read Only	The form does not have enough information to know how to set a widget within the custom control as read only. If a Control Type supports the setting of a read only state, then this property is handled at runtime.	true	true	The form applies or removes the "read-only" CSS class at the control level, and requests the ControlWrapper to refresh its rendering of the read-only state. The form notifies the ControlWrapper that the read-only state has changed by calling its `refresh()` method with the `updates` argument containing the feature name "readOnly".

Property	Description	Form Flag	Component Flag	Outcome
		true	false	The form applies or removes the "read-only" CSS class at the control level, but does not notify the ControlWrapper of changes to the read-only setting.
		false (default)	true (default)	The CSS class is not applied at the control level, but the form notifies the ControlWrapper of the change of read-only state.
		false	false	The control does not handle the read-only state.
Required	This Capability refers to the rendering of "Required" feedback. The forms framework continues to enforce that values are indeed provided when marked as required.	true	true	The form applies or removes the "required" CSS class at the control level, and also requests the ControlWrapper to refresh its rendering of the required state. This may be needed by controls that aim to provide accessibility. For example, by updating the corresponding ARIA attributes on the control widgets. The form notifies the ControlWrapper that the required state has changed by calling its `refresh()` method with the `updates` argument containing the feature name "required".
		true (default)	false (default)	The form applies or removes the "required" CSS class at the control level, but does not notify the ControlWrapper of changes to the required setting.
		false	true	CSS class is not applied at the control level, but the form notifies the ControlWrapper of the change in required state.
		false	false	The control does not handle the display of a "required" decoration.
Tab Index	For this capability, the `form` flag value is always set to `false`.	true	true	N/A
		true	false	N/A
		false	true	ControlWrapper will use the Tab Index property from the control in the generated markup for the control.

Property	Description	Form Flag	Component Flag	Outcome
		`false (default )`	`false (default)`	The control Tab Index property is not used by the ControlWrapper.

Property

The Control Type property details are as follows:

Property	Type	Description

Bindable	Boolean	If `true`, then the following features are enabled:
		
		• It is possible to bind runtime values to this property rather than just specifying static values at design time.
		
		• It is possible to update this value dynamically using script actions and that the ControlWrapper has to deal with updates to the property as notified using the `refresh()` method.
		
		• The `get()` and `set()` methods are automatically generated on the control, and content assist and script validation reflects this auto-generated API. For example: if the property name is "orientation", and it is marked as bindable=`true`, then the following two methods will be available on the control:
		
		    — `<Type> getOrientation()`
		        `setOrientation(<Type> value)`
		
		    Where `<Type>` depends on the Data Type specified on the Property.
		
		    The default value is `false`.
		
		    Note: Irrespective of the value of the bindable property, the getter and setter methods on properties are available to the ConrolWrapper through the proxy `Component.getControl()` method.

Property	Type	Description
Data Type	Classifier	Defines the type of the value for this property. This is a reference either to a built-in BOM Primitive Type, or to a Class or Enumeration defined in a BOM file in the Library project. The Data Type will determine what can be bound to the value property in the form model.
		When a property type is defined as a specific Class, then it will limit bindings of that property to objects of that type or a specialization of that type defined in the model. If the Data Type is set to BomPrimitiveTypes::Object, then the property can be bound to an object of any complex type. However, in this case it will be the responsibility of the person designing the form to ensure that the binding is to a value that can actually be used by the custom control.
		The use case for supporting complex objects for properties includes the use of complex third party controls such as tables and trees. For example, you could define a "Selection" property on a tree control that will be set to the object currently selected by the user. It would still be up the person designing the form to make sure the selected object is used correctly in the rest of the form, for example, by setting it as the value on a pane that can edit that particular type of object.
Default Value Literal	String	Provides a default value to use for this property if nothing is provided in the form model.
		The value must be a valid literal representation for the property's data type.
Externalize	Boolean	Indicates the Property provides a value which could vary based on locale. A setting of `true` here will cause the value to be externalized within the form-level resource bundle that is generated automatically.
		In addition, the property editor generated for instances of this control will expose the following two settings:
		• A property where the user can specify a value directly.
		• Allow the user to select a reference to a resource bundle key. Only one of these settings will be allowed.
		If the property is both externalized and multi-valued, then the user will only be able to specify values directly into the list editor associated with the property. These values will be written to the form-level resource bundle and can then be translated into locale-specific bundles. In this version, you will not be able to specify a resource bundle reference for multi-valued, externalized properties.
		The default value is `false`.
Label	String	Label used in the Form Designer when exposing this property in the property sheet editor.

Property	Type	Description
Multi-valued	Boolean	Indicates whether the value for this property is multi-valued. If `true`, then the value for this property can only be bound to multi-valued values. The default value is `false`.

Name	String	This name is used to expose `get()` and `set()` methods on the form Control object, and is used when providing updates to the ControlWrapper using the `refresh()` method.
		
		This property has the following restrictions:
		
		• Unique among Property names of the same Control Type.
		• Must be an NCName (that is, a legal, 'non-colonized' XML name)
		• Cannot be set to any of the names in the following restricted list:
		
		  – [n/N]ame
		  – [f/F]orm
		  – [c/C]loneIndex
		  – [c/C]ontrolType
		  – [p/P]arent
		  – [l/L]abel
		  – [s/S]hortLabel
		  – [h/H]int
		  – [e/E]nabled
		  – [v/V]isible
		  – [r/R]equired
		  – [c/C]lassName
		  – [r/R]eadOnly
		  – [b/B]ackgroundColor
		  – [f/F]ontColor
		  – [f/F]ontSize
		  – [f/F]ontName
		  – [f/F]ontWeight
		  – [v/V]isualProperty
		  – [f/F]ocus
		  – [s/S]tringValue
		  – [v/V]alue

Description	String	Provides the descriptive message to display in the status line when the property is selected in the Properties tab in Form Designer.

Property	Type	Description
Required	Boolean	Whether a value must be provided for the property. Combined with multi-valued, determines the multiplicity of the generated structural feature whose value will be set when editing the property in the Form Designer Properties tab:

Required	Multi-valued	Multiplicity
false	false	0..1
true	false	1
false	true	0..*
true	true	1..*

The multiplicity constraint is enforced by the property cell editors and form validation rules. The default value is `false`.

Control Wrapper Implementation

Each custom control needs to have an implementation of the ControlWrapper interface.

This takes the form of a JavaScript class definition that includes the methods necessary to implement the custom Control Type life cycle. The constructor is a no-argument function, with the rest of the interface implemented as function properties on the prototype for this constructor function.

initialize()

The `initialize()` method must be implemented by the ControlWrapper. It is invoked once per control instance. It is invoked after all the Control Type resource dependencies have been loaded, but before the form data model has been initialized.

Any configuration properties that are defined statically will be provided at this time, although any properties that support binding or API support may be updated after the `initialize()` method is called. The implementation needs to add the markup to the DOM at this point for the given renderMode, although there are cases when the control is being rendered in a grid pane where the markup needs to be handled in the `refresh()` method.

Method Arguments:

- **component**: an object of type Component. Component represents the form-level Control or Pane object that hosts this custom control. The form model objects obtained through the component represent read-only versions of the form models. The initial configuration of the control can be accessed using the control object in the component, including any custom properties defined by the Control Type.

- **renderMode**: This is a string that specifies the mode in which the control is to be rendered. The possible values are:

  - **normal**

  - **grid-edit**

For Control Types that specify any of the renderModes static, view-text, and view-html, those modes will not be passed into the `initialize()` method, but will instead be handled as follows:

- **static**: If the control is being rendered in a static pane, then no instance of the control wrapper is instantiated and the renderStatic() method defined on the ControlWrapper Class is called instead.

- **view-text** and **view-html**: The form will access the getFormattedValue() method of the ControlWrapper when a view-only version of the control is needed.

Once a control instance has been asked to render in a particular mode, that instance will not be asked to render in a different mode.

refresh()

This method must be implemented by the ControlWrapper. It is invoked for rendering of the control in the same Render Mode as originally specified in `initialize()`.

This method is only called after the `initialize()` method, and is called at any point when the control configuration or value has been updated. This method will be called at least once after initializing.

Functionality for Grid Panes

For Control Type instances that are being rendered in a grid pane, this method is called once each time a different cell in the grid pane is edited. In those cases, it is not always necessary to regenerate the entire DOM structure of the control. You can update the existing DOM structure, which was previously rendered, based on any updates to the configuration or value of the control. This is applicable only if the control is not in the **Always Render** mode. The `getControl()` method on the component interface at this point returns the control instance for the grid pane cell, which is being edited.

Method Argument:

- **updates**: This is an array that contains the names of configuration properties updated since the last `initialize()` or `refresh()` method invocation. For example: if the array contains the value `myProperty`, then that means the value of the custom property named `myProperty` has been updated since the last `refresh()`. The full set of configuration properties can always be accessed using the control in the component object passed to the ControlWrapper in the `intialize()` method. There is a set of built-in keys that can reference properties common to all controls: "label", "hint", "required", "enabled", "readOnly", "visible", "locale", and "validation". The updates array can also contain the custom property names, if the value of any of those properties changed since the last `refresh()` method call.

> When the control is in a grid pane that is not in **Always Render** mode, the ControlWrapper is shared among the cells of a column. The `refresh()` method is not called immediately after a property is changed on a cell but only when that cell is edited. The `updates` array in such cases is empty. The ControlWrapper implementation must query the specific control instance for any change in property values. You can retrieve the specific control instance by using the `getControl()` method on the component interface.

destroy()

This method must be implemented by the ControlWrapper. This method is invoked when the control instance is being taken out of service.

getValue()

This method returns the value modified or rendered by this control. For complex types, this is the JavaScript BOM object that corresponds to the instance type.

Method Return value:

- Returns the control value

getFormattedValue()

This is an optional method that returns a simple read-only rendering of the value managed by this control. This method only needs to be implemented if either the **view-text** or **view-html** render modes are supported. At most, one of these modes can be supported.

- **view-text**: The return value of this method will be plain text that is rendered in the DOM within a DOM `Text mode`.

- **view-html**: The return value is a string representation of HTML markup and is treated as such when added to the DOM.

See the `com.tibco.forms.extension` package for a set of built-in utilities for formatting values of various types.

Method Argument:

- **value**: This is the value to be formatted.

Method Return Value:

- Returns the formatted control value as text or html.

isReady()

This method returns `true` if the ControlWrapper is ready to be initialized. This method is repeatedly called until it returns `true` or the loading of the form times out.

This gives the wrapper a chance to check whether necessary libraries are loaded prior to initialization. If only the needed libraries are specified directly in the Components Library model, then it should be always safe to return `true` from this method. However, some frameworks, such as GWT and Dojo, will load additional files that are not loaded directly by the Forms framework. For these cases, the wrapper should perform a check. For example: by checking for the existence of a needed function or class, before returning `true`.

This must be a static method on the ControlWrapper.

Method Return Value:

Returns a boolean value.

setFocus()

This is an optional method that only needs to be implemented if the "focus" capability for the Control Type is set to "component". This method sets the focus for this control. Use in situations where the focus is explicitly set using the API for this control.

If the `setfocus()` method is defined in the ControlWrapper, the capability always picks the "focus" capability from the ControlWrapper. If you do not want the component to handle `setFocus()` then do not define it in ControlWrapper.

compare()

This method is optional.

It only has to be implemented at the class level (not the prototype level) for ControlWrappers that meet both of the following conditions:

- The value edited by the control is of a complex type, i.e. the type is specified by a BOM class.
- Instances of the control are allowed to occur in grid panes.

In these situations, if the grid pane has enabled sorting, the form needs to know how the complex objects should be sorted. The compare method is used in performing this sorting.

Method Arguments:

- **value1**: This is the first object to compare.
- **value2**: This is the second object to compare.

Method Return Value:

- Returns an integer value:

    - < 0: if **value1** is less than **value2**
    - 0: if **value1** is equal to **value2**
    - > 0: if **value1** is greater than **value2**

renderStatic()

This static method is optional, and only has to be provided for Control Types that support the "static" Render Mode. This method is invoked whenever the form needs to obtain a static rendering of the value bound to this control.

This must be a static method on the ControlWrapper, as an instance of the ControlWrapper is not instantiated when the control is located in a static pane.

The value returned by the control is added as HTML to the form. Any user input values should be escaped appropriately to avoid them being interpreted as HTML.

Method Arguments:

- **value**: This is the value that needs to be formatted.
- **label**: (Type String) - The label to be rendered for the static control.
- **hint**: (Type String) - The hint to be rendered for the static control.
- **labelId**: (Type String) - Identifier of the label as rendered by the form. This is useful for accessibility.
- **propertySet**: (Type Associative Array) - Initial configuration of the control, including custom properties configured in the Form Designer. The key is the name of the property as defined in the Control Type.
- **resource**: (Type Object) - The same as retrieved from the `Component.getResources()` method.
- **textOnly**: (Type Boolean) - If `true`, then the pane this is being rendered in is expecting a **text-only** rendering of the value. That is, no rendering of a widget that displays the value.
- **parentPaneType**: (Type String) - This is the string that represents the type of pane. This is equal to the value returned from the `Pane.getPaneType()` method. A ControlWrapper identifies the rendering on a grid pane using the `parentPaneType` argument.
- **logger**: (Type logger Object) - This is the same logger object available in Form action scripts. The logger object helps to log messages to the form log. View the form log at preview time by using the appropriate logging level in the **Windows - > Preferences - > Form Designer - > Preview** page. Enable logging at runtime by using the query parameter `log_level` with an appropriate value: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. For example: http://<server>:<port>/openspace? log_level=INFO. See the table Logger Class for the list of available methods.

Method Return Value:

- Returns an HTML string.

Component Interface

The `initialize()` method for the ControlWrapper receives an object of type Component. This provides an interface that the ControlWrapper can use to obtain information and configuration from

the form layer, and to also raise events back to the form so they can trigger rules defined in the form model.

The Component object provides the following APIs:

generateId()

This method generates an identifier that is unique on this page. It allocates IDs to the HTML Elements created within the ControlWrapper.

If the ID returned by this method is not used by the wrapper, the next time it will return the same ID as it is still unique with in the document. As a convenience, if a suffix is provided as an argument to this method, the returned value will have the suffix appended. If no suffix is provided, then the base identifier will be returned.

A ControlWrapper would want to use this method if it generates any DOM nodes in the HTML that need to be directly referenced using ID. Using this method will ensure that the ID used will not be in conflict with other IDs on the page.

Method Argument:

- **suffix** (optional)

Method Return Value:

- unique ID

getControl()

This method returns the Form-level Control that corresponds the custom control instance. This provides access to all the getter methods that are available on the control object in a form-level action script. It also provides the getter and setter methods for all the custom properties.

Method Return Value:

- Control

getFactory()

This method returns the factory object that is available within Form action scripts. This will allow wrappers to create new objects as part of their functionality. This object will expose factories that are in the library project as well as those available from the Forms project using the custom control and any in referenced projects, recursively.

Method Return Value:

- Object

getForm()

This method returns the Form that contains the custom component.

Method Return Value:

- Form

getHintId()

This method returns the ID of the DOM node which renders the standard, form-supplied hint for this control. It is useful in situations where the hint needs to be referenced for accessibility purposes. For example: by using the ID in corresponding ARIA attributes on the control widgets.

Method Return Value:

- String

getLabelId()

This method returns the ID of the DOM node which renders the standard, form-supplied label for this control. It is useful in situations where the label needs to be referenced for accessibility purposes. For example: by using the ID in corresponding ARIA attributes on the control widgets.

Method Return Value:

- String

getLocale()

This method returns the String representation of the locale in which the control should be rendered.

Method Return Value:

- String

getParentNode()

This method returns the parent DOM node into which this control should render its contents.

Method Return Value:

- DOM Element Node

getPresentationURL()

This method returns the base URL of the Presentation Resources folder of the project, in which a custom control is defined.

For example, if a project contains `myimage.gif` located at `project/Presentation Resources/images`, you can compute the URL of this image using:

```
var mypath = component.getPresentationURL() + "/images/myimage.gif";
```

Method Return Value:

- The base URL without a trailing path separator

getResources()

This method returns an object that provides access to all localized resource bundles defined at the library level and the component level for this particular Component.

Resources are accessed using the resource name and individual keys. For example: for a resource with name `resource.myName` created at the library or at the component level that has a key called `myLabelKey` in it, its value can be retrieved using:

```
var resources = component.getResources();
    var myLabel = resources.myName.myLabelKey;
```

The resources returned correspond to the locale in effect for the form when rendered.

Method Return Value:

- Object

getValidationMessageIds()

This method returns an array of DOM identifiers which represents any validation messages currently in effect for this control. There is one ID for each message pane in the form. Useful in situations where the messages need to be referenced for accessibility purposes.

If null, then there are currently no validation errors reported against this control. If the array is non-null but empty, this signifies that there are errors, but no messages are displayed because the Form does not contain a Messages pane.

Method Return Value:

- String

raiseEvent()

This method is invoked by the ControlWrapper when it needs to propagate an event back to the Form layer. Most controls should raise at least the **update** event in order to notify the form layer that the control value has changed. It is not necessary to raise an update event when updating the attribute value of a complex object or updating the list for a multi-valued complex type. The BOM JavaScript representations of these objects handle the updates internally.

Method Arguments:

- **eventName**: Name of the event as configured in the component metadata. This name should correspond to one of the events specified as supported by the component type. Built-in events include **close, doubleclick, enter, exit, localize, open, select, submit, and update.**

- **eventValue**: Object that differs depending on the event being raised. For **update** events, this is the new value. Other events do not need an eventValue. Any custom-defined events ignore the eventValue.

BOM JavaScript API for Custom Controls

In order to support more complex custom control use cases, the JavaScript API documented for the auto-generated BOM JavaScript classes has been augmented to provide a reflective API.

This allows controls to write code that can introspect objects and provide an auto-generated UI based on the type of object. For example, this would enable implementation of controls such as a tree control that can provide a user interface based on the structure of arbitrary objects. The API described here is currently only supported for use within Custom Controls.

Factory Methods

The methods listed in the table Factory Methods are available in the factory that is available for each BOM package.

Factory Methods

Method	Return Value	Description
create(className) • className is a fully-qualified name of the BOM JavaScript class. This must be a class managed by the given factory.	Object	Creates an instance of the given class. ControlWrapper uses this method to support cases where the type of object being managed by a complex custom control is not known at design time. From form action methods, the specific createXXX() method for a given class should be used.
getClass(className) • className is a fully-qualified name of the BOM JavaScript class. This must be a class managed by the given factory.	Object	Returns the class object for the class with the given name.

BOM Class Methods

The methods listed in the table BOM Class Methods are available on the class object returned from the getClass(className) method of a factory, or the getClass() method of a BOM JavaScript object instance.

BOM Class Methods

Method	Return Value	Description
getAttributeNames()	String[]	Returns a JavaScript string array containing the names of all attributes for this class. These are names as defined in the BOM for this class and all of its super-classes. For complex children, these will correspond to the name of the association endpoint for the child. This array should not be modified. This array is the union of attribute names retrieved using getPrimitiveAttributeNames() and getComplexAttributeNames().
getPrimitiveAttribute Names()	String[]	Returns a JavaScript string array. These are attributes with simple data types; i.e., primitive types and enumerations. These are names as defined in the BOM for this class and all of its super-classes. This includes both single- and multi-valued attributes. This array should not be modified.

Method	Return Value	Description
`getComplexAttribute Names()`	String[]	Returns a JavaScript string array containing the names of all complex children of this class. These are names of the association endpoints for these children as defined in the BOM for this class and all of its super-classes. This includes both single- and multi-valued attributes. This array should not be modified.
`getAttributeType (attName)` • `attName`: name of attribute.	String	Returns the type for given attribute. This will either be the fully-qualified class name as defined in the BOM if the attribute is complex, or will be one of the following primitive types: • BomPrimitiveTypes.Boolean • BomPrimitiveTypes.Dates • BomPrimitiveTypes.DateTime • BomPrimitiveTypes.DateTimeTZ • BomPrimitiveTypes.Decimal • BomPrimitiveTypes.Duration • BomPrimitiveTypes.ID • BomPrimitiveTypes.Integer • BomPrimitiveTypes.Text • BomPrimitiveTypes.Time • BomPrimitiveTypes.URI
`isAttributeMultivalued (attName)` • `attName`: name of attribute.	Boolean	Returns `true` if the attribute with the given name is a multi-valued attribute as defined in the BOM.
`isAttributePrimitive (attName)` • `attName`: name of attribute.	Boolean	Returns `true` if the attribute with the given name is of a primitive type or enumeration. If it returns `true`, it will be a member of the array returned from `getPrimitiveAttributeNames()`.

BOM Class Instance Methods

The methods listed in the table BOM Class Instance Methods are available on each instance of a BOM JavaScript class.

BOM Class Instance Methods

Method	Return Value	Description
`getAttributeValue (attName)` • `attName`: name of attribute.	Object	Returns the value of the attribute with the given name. The return type depends on the type of attribute being retrieved. It will be one of the following: • `attName` is primitive and single-valued: returns a value of type String, Number, Date, or Duration, depending on the specific type of attribute. • `attName` is primitive and multi-valued: returns a JavaScript array containing the underlying values. • `attName` is complex and single-valued: returns an instance of the BOM JavaScript class associated with the attribute. • `attName` is complex and multi-valued: returns a List object containing the underlying values. The method throws an exception if the underlying class does not support an attribute with the given name.
`setAttributeValue (attName, value)` • `attName`: name of attribute. • `value`: new value of attribute.	Void	Sets the value associated with the given attribute name. The type of object depends on the attribute being set: • `attName` is primitive and single-valued: should be a value of type String, Number, Date, or Duration, depending on the specific type of attribute. • `attName` is primitive and multi-valued: should be a JavaScript array containing the underlying values. • `attName` is complex and single-valued: should be an instance of the BOM JavaScript class associated with the attribute. • `attName` is complex and multi-valued: unsupported. In this case, the List object obtained from the object should be updated directly with additions or deletions. The method throws an exception in the following scenarios: • if the underlying class does not support an attribute with the given name. • if an attempt is made to set the value of a complex multi-valued attribute.

Utility Methods

A set of JavaScript methods are provided by the forms framework to aid custom control developers. These are for use by the custom control wrappers.

See Utility Methods for the complete list of API methods.

Reference

In order to work with TIBCO Business Studio Forms, you must be aware of various details of the modeling environment.

You must know and understand the controls that can be placed on a form, the properties associated with each type of control, the validation errors that can appear in the Problems view, and the use of scripts to extend the functionality of your forms.

The Workbench

The Eclipse workbench for modeling forms appears by default when TIBCO Business Studio is installed. The BPM Modeling perspective provided by TIBCO Business Studio includes certain views and editors that are important for designing and deploying forms.

In Eclipse, the term *view* refers to an area of the workbench that displays information related to your Eclipse projects. In the BPM Modeling perspective of TIBCO Business Studio, for instance, there are a number of views, such as the Project Explorer view, the Properties view, and the Outline view, that display objects and information in support of the modeling work you perform in the Process Editor or Form Designer.

Eclipse Workbench with Project Claims Process - No Forms
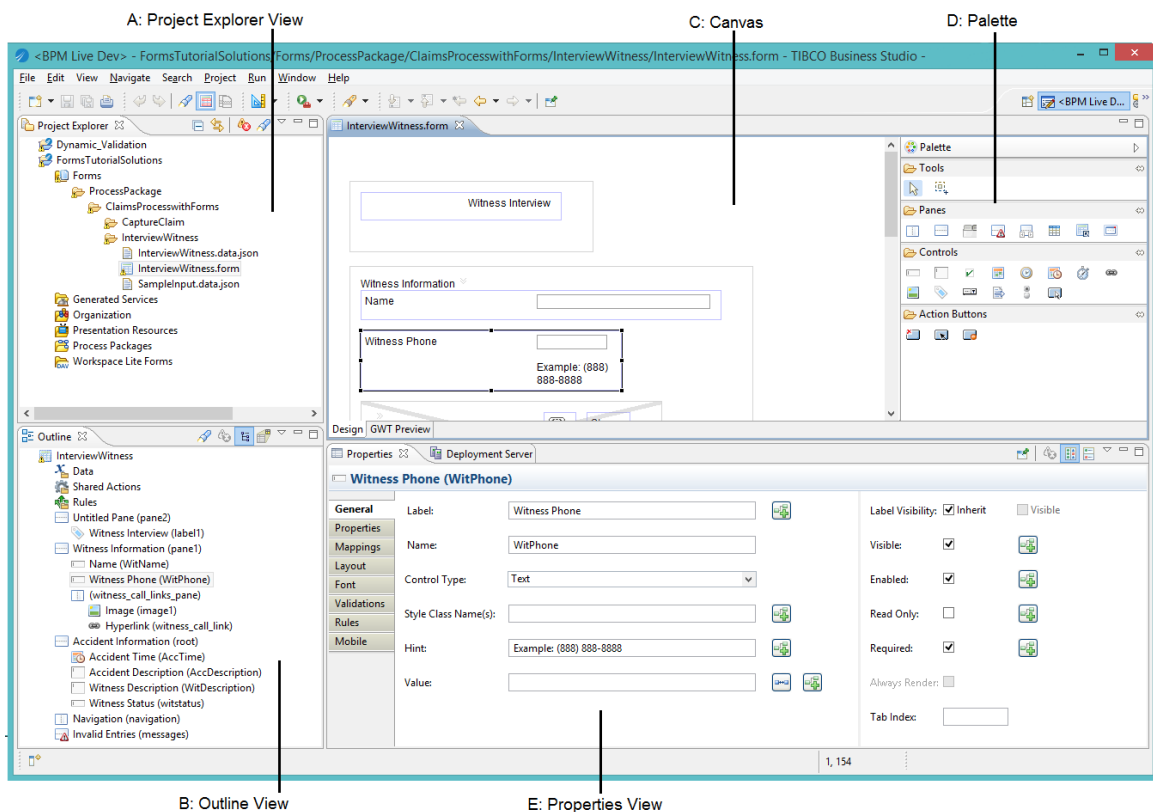
- **A: Project Explorer**

  The Project Explorer view shows your TIBCO Business Studio projects and all project resources, including Process Packages, Business Assets, and Forms, arranged in hierarchical tree structures.

- **B: Outline**

  The Outline view shows non-visual and visual elements of the form including form parameters, shared actions, rules, controls and panes.

For more details, see Outline View and Outline View Context Menu.

- **C: Canvas**

  The Canvas is where you create your forms. On creating a form, you notice two tabs at the bottom: a **Design** tab for modeling forms, and **GWT Preview** tab for viewing and testing the forms.

  For more details on working in the modeling or preview mode, see The Design Tab and Preview Tabs and Form Designer Canvas Context Menu.

- **D: Palette**

  The palette is a part of the Form Designer and provides tools for adding panes and controls to a form, and for selecting objects on a form. Click the **Palette** arrowhead in the upper right corner to open the palette. The arrow is a toggle between a visible and a hidden palette.

  There is also the detachable Palette view (**Window** > **Show View...** > **Palette**). This palette is very useful to save space when working on multiple processes and/or forms.

  For more details, see The Palette for the Form Designer

- **E: Properties**

  The Properties view shows the properties of a selected object on a form, such as a pane or a control, and allows you to edit the values of those properties.

  For more details, see

  - Properties View Tabs
  - Properties View for Forms
  - Properties View for Panes
  - General Tab

The Palette for the Form Designer

To create a form, you need to add parameters to a user task from the existing process data fields.

Right-clicking the user task, and selecting **Forms** > **Open** , opens the relevant form. You can modify the form by adding or moving panes and controls using the tools on the palette.

To add a pane or control, click the tool on the palette for a specific object, position your mouse pointer in the appropriate location on the Form Designer, and click to place the object on the form. (To *move* a pane or control already on the form, use either the drag-and-drop or cut-and-paste techniques.)

When you hover the mouse over the icons in the palette, a pop-up tool tip describes the tool indicated by the mouse point. The palette contains tools as described in the table Form Designer Palette.

Form Designer Palette

Palette Item	Description
Select	Allows you to select objects.
Marquee	Allows you to select several objects by drawing a box around them.
Vertical Pane	Adds a Vertical Pane , whose children are arranged vertically.

Palette Item	Description
Horizontal Pane	Adds a Horizontal Pane , whose children are arranged horizontally.
Tabbed Pane	Adds a Tabbed Pane , whose sub-panes are represented as clickable tabs.
Message Pane	Adds a Message Pane for displaying error messages.
Record Pane	Adds a Record Pane to edit a list of complex objects, one record at a time.
Grid Pane	Adds a Grid Pane to work with list of complex objects.
Embedded Form	Allows you to embed another form within the parent form.
Text	Adds a Text control for typing a single line of text.
Text Area	Adds a Textarea control for typing multiple lines of text.
CheckBox	Adds a Checkbox control for making a binary (yes or no) choice.
Date	Adds a Date control for specifying or picking a calendar date.
Time	Adds a Time control for specifying or picking a time.
Date-Time	Adds a Date-Time control for specifying or picking a calendar date and time.
Duration	Add a Duration control for specifying duration using a configurable set of units.
Hyperlink	Adds a Hyperlink control that renders a clickable hyperlink.
Image	Adds an Image control that renders an image.
Label	Adds a Label control that renders a non-editable label.

Palette Item	Description
Option List Optionlist	Adds an Optionlist control for picking from a list of options.
Pass-through Pass-through	Adds a Pass-through control that renders HTML markup.
Radiogroup	Adds a Radiogroup control for picking one of a set of mutually exclusive options.
Button Button	Adds a Button control to the form.
Cancel Button Cancel Button	Adds a Cancel Button for discarding the changes and closing the form.
Submit Button Submit Button	Adds a Submit Button for submitting the changes and closing the form.
Close Button Close Button	Adds a Close Button for applying the changes and closing the form.

The Palette View

To expand the palette, hover the button over **Palette** to the right of the Form Designer. To add a pane or a control on the form, click the appropriate button. After adding the controls, the palette collapses to its original state automatically.

Another way to keep the palette from taking up extra space in the Form Designer is to use the Palette view, which opens the palette as an ordinary Eclipse view. When doing this, the Palette view appears by default as a tab along with the Properties view and Problems tabs, but it can be dragged to other locations. The Palette view is then shared between all open graphical editor instances, hiding local fly-out palettes in any open graphical editors.

Open the Palette view by clicking **Window** > **Show View** > **Palette** . If you close the Palette view (by clicking its close box), the fly-out palette returns for the graphical editor instances where it was previously displayed.

The arrow now points rightward. When expanded by this method, the palette remains visible (as the Palette view) until the arrow is clicked again.

Panes

Panes serve as containers for controls or for other panes, and provide a means of controlling the visual layout of objects on a form.

Similar to controls, panes have attributes such as a label, background color, and visibility. The child properties of a pane define the arrangement and display of the controls in the pane.

Vertical, Horizontal, Tabbed, and Message Panes

Nesting Panes

Panes can be nested inside other panes for greater flexibility in the positioning of controls. For instance, you can place two vertical panes within a single horizontal pane. This results in a two-column layout of controls for the portion of the form defined by the horizontal (parent) pane.

All types of panes, except for tabbed panes, can be nested to create tabs. Panes can also be rearranged by dragging and dropping within the form Outline view.

It is strongly encouraged to avoid nesting panes to an extreme number of levels, since this complicates the form model and can affect performance.

Nested panes can be used to arrange controls on the form. For instance, you can create a two-column layout by adding a horizontal pane to the canvas, and then nesting two vertical panes within it. The

same approach can be used to create additional columns: just place additional vertical panes inside the original horizontal pane.

There are two restrictions on the nesting of panes:
- No other pane can be nested within a message pane
- A tabbed pane cannot be nested in another tabbed pane

Types of Panes

There are several types of panes: vertical, horizontal, tabbed, message, grid, record, embedded form, and modal dialog panes. Each pane type is represented by an icon in the palette.

It can be difficult to distinguish between a vertical pane and a horizontal pane before you place any controls or child panes in them. For this reason, these pane types are distinguished in the **Design** tab by small chevron icons pointing down for vertical panes and to the right for horizontal panes. (The chevrons do not appear at runtime or in the preview tabs.)

Design View
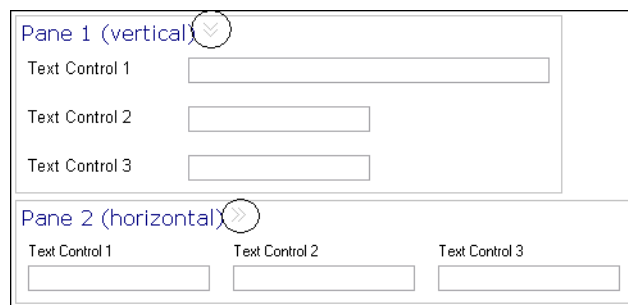
Special considerations apply for resizing tabbed panes when additional sub-panes are added. For more information, see Specialized Layouts.

Vertical Pane

A vertical pane is a pane in which controls are arranged vertically, with one above the other. Vertical panes are auto-sized to hold controls, child panes, or both.

Horizontal Pane

A horizontal pane is a pane in which controls or child panes are arranged horizontally, with one next to the other. Horizontal panes are auto-sized to hold the controls or child panes within them.

Tabbed Pane

Tabbed panes provide a means of stacking a set of related panes such that one pane at a time is visible. Each pane has a corresponding tab, which are arranged in sequence along one of the tabbed pane's vertical or horizontal edges.

Keyboard Access

Change between tabs without a mouse by using the left and right arrows.

The direct children of a tabbed pane must be panes, not controls. The canvas prevents you from inadvertently placing controls directly inside a tabbed pane. Clicking on or otherwise selecting a tab activates its associated pane (make it visible). This "select-tab-to-activate-pane" behavior is common to both design time and runtime. At design time, however, a tabbed pane offers additional capabilities to aid in the design process:

- **Add a new child pane**

A special button positioned at the end of the tab collection. Click this button to add a new child pane to the tabbed pane.

- **Expand/collapse the tabbed pane**

   A special toggle button positioned after the new pane button, that toggles the state of the tabbed pane between a collapsed state and an expanded state. The collapsed state has one pane visible whereas the expanded state has all child panes visible side-by-side. In the expanded state, the tabbed pane behaves similar to a horizontal or vertical pane. You can add, move, and delete controls on the expanded pane. The expanded state is particularly useful when you want to rearrange or delete child panes or move controls between panes.

   > While adding controls to tabbed panes, keep the pane expanded.

Message Pane

A message pane is used to display validation error messages. Message panes cannot contain panes or controls. A message pane displays the message typed in the Message field of a control's Define Validation dialog if the validation script in the Script field returns a value of `false`.

> If a message pane is added to a modal dialog pane, it shows only the validations that are applicable on the modal dialog pane.

The figure Script and Message Example for a Message Pane shows a typical validation script and message for a Text control.

Script and Message Example for a Message Pane


```
Script:   this.getForm().isNumber(this.getValue(),3);

Message:  Customer Age should be a number
```

The appearance of a message pane, the label font and layout, can be configured through the Properties view for the pane.

Record Pane

A record pane is used to edit a list of complex objects, one record at a time. It supports the ability to view and edit the contents of one element of an array of complex objects. A set of navigation controls is provided to support moving between the records in the list. The record pane uses the same layout as the vertical pane.

The record pane displays the contents of a list of objects. The contents of the list in a record pane is linked with the list in one of the following ways:

- The pane value is bound to a multi-valued complex parameter.
- The pane value is bound to a multi-valued child of a complex object.
- The value of the pane is updated with a list of complex objects via an API function call.
- The value of the pane is updated with a list of complex objects via a computation action.

For either of the latter two approaches, the **Pane Data Type** property needs to be set in the form model to the type of object that is set on the pane with a list.

The **Properties** tab of a record pane's **Properties** view displays a set of properties. You can refer to Properties View for Panes section for the complete listing of the **Properties** tab.

Navigation Controls

The controls on the navigation bar perform the following operations:

- Go to the first record.

- Go to the previous record. There is no change if you are at the first record.

- Go to the next record. There is no change if you are at last record.

- Go to the last record.

The label displays: `Record [index] of ##`. The `index` indicates the current record and `##` indicates the length of the list. You can directly edit the `index` value. If a number less than 1 or a number greater than the length of the list is specified, the index is reset to the value it was set earlier.

Grid Pane

Grid panes are generated in a default form for complex data when the data type is defined as allowing *multiple* instances, for example, zero-to-many (*) or one-to-many (1..*). When a grid pane is rendered, attributes of complex data types correspond to columns, and each of the multiple instances corresponds to a row.

By default, the data displayed in a grid pane is not sorted by columns. If the default sort column and sort order are specified, the form is loaded initially according to the specified sorting. At runtime, clicking the column header sorts the rows in ascending order based on the values in that column, and clicking the column again sorts the rows in descending order. Clicking once more on the column restores the unsorted natural record order.

Sorting data in a grid pane is case insensitive. Numbers in a text control are sorted as text, and not as numeric data. To sort numbers in numerical order, use the numeric control.

Several properties appear in the **Properties** tab of a grid pane's **Properties** view that are particular to this type of pane. Refer to Properties View for Panes section for the complete listing of the **Properties** tab.

Modal Dialog Pane

A modal dialog pane blocks the rest of the form until the user closes the modal dialog.

This pane has a title bar and an optional **Close** button. However, by default a **Close** (**X**) button is displayed on the title bar. You can configure **OK** and **Cancel** buttons as required. By default, the pane also supports the Escape key.

Vertical or horizontal panes can be converted into modal dialog panes. You cannot add a modal dialog pane to another modal dialog pane, nor to a record pane, nor as a direct child (tab) of a tabbed pane. A modal dialog pane cannot be static.

Modal dialog panes cannot be marked for deferred rendering or deferred initialization. However, a child pane of a modal dialog pane can be marked for deferred rendering or for deferred initialization.

Modal dialog panes support the following events:

- **open**

  Fired when the dialog is opened, that is, made visible

- **close**

  Fired when the dialog is closed using the explicit **Close** button

- **cancel**

  Fired when the dialog is closed using either the close button (**X**) on the title bar, or using the **Cancel** button, ignoring the changes made by the user on the dialog

Setting Pane Properties with Bindings and Rules

To associate pane properties such as Label, Visibility, and Enabled with the values of controls or parameters, you can use bindings and rules.

See the following:

- Bindings
- Rules

Controls

Controls are the objects on a form that take your input, such as text fields, check boxes, and radio buttons. Controls must be placed within a pane. Controls can be rearranged within the form by dragging-and-dropping them.

Controls can be copied and pasted within a form or between forms. It is also possible to re-arrange the position of controls by dragging the controls within the form Outline view. Controls always include their associated labels, although these can be hidden.

Text

The Text control allows users to type text. You can make this control read-only.

Textarea

The Textarea control allows users to type multiple lines of text. The length, in number of characters, and the number of lines are configurable. The default values are 25 characters and 10 lines.

You can make this control read-only.

Checkbox

Checkbox controls represent Boolean values. In effect, they are on/off switches that may be used as a toggle. The switch is on (that is, the Boolean value is true) when the check box is selected. By default, a Boolean IN/OUT data parameter is displayed as a check box. If the data parameter is mandatory, it means the check box must be selected, otherwise a form validation rule prevents the form from being submitted. Use this feature when you want the user to acknowledge that some aspect of the form has been checked or considered. If you need the user to select true or false with the check box, the data parameter must be optional rather than mandatory.

Date

The Date control allows users to specify date, either in a string in the correct format, or by clicking a calendar widget. The calendar opens when the user clicks the icon next to the Date control's text field:
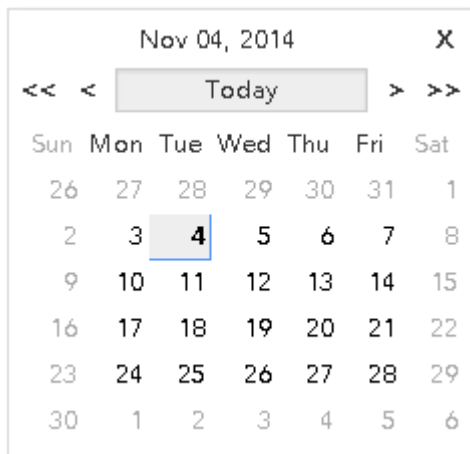
You can make this control read-only.

Time

The Time control has up and down arrows for incrementing the hour or minute value, or for toggling between AM and PM. The value that is changed by clicking the arrow depends on whether the cursor is in the hour field, the minute field, or the AM/PM field. You can also specify a time by keying in a string in the correct format.

You can make this control read-only.

Date-Time

The Date-Time control allows users to input a date and time. The date portion of the control has a calendar widget for selecting a date with the mouse. The time portion has up and down arrows for incrementing the hour or minute value, or for toggling between AM and PM. The value that is changed by clicking the arrow depends on whether the cursor is in the hour field, the minute field, or the AM/PM field. You can also specify a date and time by keying in a string in the correct format.

You can make this control read-only.

Duration

The Duration control allows the users to specify a duration using a configurable set of units. The **Properties** tab of the **Properties** view for the Duration control allows you to select which units are displayed for parameters of type Duration. The supported units are:

• Years

• Months

• Days

• Hours

• Minutes

• Seconds

• Milliseconds

Any combination of these units can be enabled.

You can make this control read-only.

To avoid any loss of information throughout a process, it is best to edit Duration values using the same set of units in all forms that modifies the underlying Duration parameter. This holds as well for scripts that create or modify Duration objects.

Hyperlink

The Hyperlink control allows the Form Designer to place a clickable hyperlink on the form.

Image

The Image control allows the Form Designer to place an image on the form, referenced by a URL.

Label

The Label control allows you to display static text that the user cannot edit. The label control still has its own label field that is used to identify the value being rendered by the label control.

Optionlist

The Optionlist control allows the Form Designer to create a drop-down list.

Pass-through

The Pass-through control is a widget that allows users to specify a block of arbitrary HTML into a form. Specify the HTML fragment in the large editing box on the **Properties** property sheet. The markup is inserted directly into the browser DOM at runtime.

The binding dialog allows you to set the markup via a binding or computation action, just as with other form values. (Bindings allow you to tie the value of an item, such as a control, to the value of something else in the form, without coding.) You do not need to configure bindings or computation actions, however, in order for the Pass-through control to work.

Radiogroup

The Radiogroup control allows users to choose among the listed options. Only one option (one radio button) can be selected at one time.

Button

There are various kinds of buttons on the palette: **Button**, **Cancel Button**, **Submit Button**, and **Close Button**.

When one of the Cancel, Submit, or Close buttons is added to the canvas, an associated Rule is also added to the Form to handle the select event on that button. A generic Button that is added must be configured to invoke an action when clicked.

The Properties views are identical for each of them, except that the default value that is selected in the **Button Type** list on the **Properties** tab is **primary** for the **Submit Button**, and it is **peripheral** for the other button types. This means that a **Submit Button** that is placed on a form from the palette, by default, is invoked on a mouse click or when you press the **Enter** key. (If the focus is within a textarea control or similar control, the **Enter** key is interpreted by the local component. It may not invoke the **Submit** button.) Buttons of the other types are invoked only when they are clicked. Primary buttons are distinguished with a dark single-pixel border. A form may contain at most one Primary button.

In addition to primary or peripheral, the value of the **Button Type** list can be set to **associative**.

An **associative** button is one that is associated with another control. For example, an associative button called `browse` might be located next to a file upload control.

Edit as List with a Control

The List control is not an independent control in itself, but is a special control property that can be enabled for a Text, Text Area, Date, Time, Date-Time, or Duration control to represent multiple instances of primitive data.

It allows you to add and delete items, or move them up and down. The list functionality is enabled with the check box **Edit as List** on the **Properties** tab of the **Properties** view for controls that support list editing.

A primitive attribute in the business object model that has a multiplicity of * (zero or more instances are allowed) corresponds to an array, and is represented on the default generated form with the **Edit as List** property enabled.

Likewise, a control with the **Edit as List** property enabled is rendered for a primitive process **Data Field** with the **Array** check box selected.

Control or Component Labels

You can give a label to each control or component. Sensible labels help users understand the control or the component easily. Labels are shown together with the names of the controls.

Users in the Business Analysis mode cannot see the physical name, which is used only by the users in scripts and is visible in property editors.

Consistent Use of Labels

The same labels must be used through the Forms Designer UI for consistency, including in the Outline view, pickers, wizards, and property views.

Labels for Rules and Actions

Rules and Actions show the label description even if they are long. For example,

```
Guardian required for underage drivers (guardian_required)
Get employee details (svc_empdetails)
```

Using the Option Include Type Name in Labels

When using this option, search expression should start with an asterisk (*) if the text you're trying to match is not at the beginning of the label. This is applicable to the various picker UIs such as the Binding picker, Event picker, and so on.

For example, when you are binding the value of a control to synchronize with a parameter that has a name **CustPhone** and a label **Customer Phone,** and if you want to search that parameter with a keyword **Phone**, type the keyword expression as ***Phone**. This shows all the items that have the text **Phone** in their labels.

Similarly, if you are a user with the Solution Design capability and you want to search the parameter by name, you need to type the keyword expression as ***CustPhone**.

Properties View Tabs

Forms, panes, and controls can be configured by specifying or modifying values in Properties views. Each form, as well as each of its panes and controls, has a Properties view with a set of tabs, and each tab provides access to a group of properties.

The tabs on a Properties view provide easy access to the many properties you can set for the objects on a form. Properties tabs are provided for: Forms, Data, Parameter, Shared Actions, Rules, Pane, and Controls.

Properties View Tabs

Properties View Tabs	Description
Forms	
General Tab	Specify a CSS class to be used for styling at the form level.
Mappings Tab	Shows a global view of all the bindings and computation actions in the form.
Font Tab	Settings for font properties at the form level, which may be inherited by objects on the form.
Child Layout Tab	Settings for layout properties of top-level panes, which inherit from the form.
Child Labels Tab	Setting for label properties of top-level panes, which inherit from the form.
Rules Tab	Shows the rules to be triggered by a form event.
Resources Tab	Shows resources associated with the form, such as JavaScript files and images.
Preview Data Tab	Setting for a preview data file, either none (no data appear initially for the controls), default, or custom (to assign a custom preview data file to the form).
Panes	
General Tab	General properties of the pane.
Properties Tab	Visual properties of the pane, inherited from the containing pane or form by default which, in turn, overrides the system defaults.
Mappings Tab	Shows a global view of all the bindings and computation actions related to the pane.
Layout Tab	Layout properties for the pane, inherited from the parent pane by default.
Font Tab	Font settings, used if the Form Designer does not want the pane to inherit these properties from the containing parent form or pane.
Child Layout Tab	Settings for layout properties of those objects that inherit from this pane.
Child Labels	Setting for label properties of those objects that inherit from this pane.
Validations Tab	For writing scripts that run when the form is submitted or updated, and to check whether you have provided valid input for the pane. This tab is not visible when in Business Analysis mode.

Properties View Tabs	Description
Rules Tab	The Rules tab lists the Rules triggered by each of the events supported by the pane, and provides a mechanism to create new Rules for that the pane.
Mobile Tab	Used for mobile specific configurations.
Controls	
General Tab	General properties of a control.
Properties Tab	Properties that are specific to the type of the control being configured. Fields on this tab vary between control types. Some control types do not have a properties tab.
Mappings Tab	Shows a global view of all the bindings and computation actions related to a control.
Layout Tab	Layout properties for the control, inherited from the parent pane by default.
Font Tab	Font properties for the control, inherited from the parent pane by default.
Validations Tab	For writing scripts that run when the form is submitted or updated, and to check whether you have provided valid input for the control. This tab is not visible when in Business Analysis mode.
Rules Tab	The Rules tab lists the Rules triggered by each of the events supported by the Control, and provides a mechanism to create new Rules for that control.
Mobile Tab	The Mobile tab is used for mobile specific configurations.

Properties View for Forms

The Properties view for a form contains eight tabs: General, Mappings, Font, Child Layout, Child Labels, Rules, Resources, and Preview Data. The form Properties view can be found by selecting the root-most element in the Outline view or clicking outside the panes of the form.

General Tab

The setting in this tab is used to specify one or more CSS classes for styling the form.

Fields on the Forms General Tab

Property	Description
Style Class Name or Names	Field for indicating the name of a CSS class within a CSS file that has been associated with the form. The CSS class is used for styling at the form level.

Mappings Tab

This tab is used to view, edit, and create mappings for the form. You can refer to Mappings Tab section for further details.

Font Tab

Fields on the Forms Font Tab

Property	Description
Inherit from System Defaults	Check box determines whether or not the values on this tab are inherited from the system defaults.
Font Name	Determines the default font used to render control text and hints throughout the form.
Font Size	Determines the default height (in points) of the font used to render control text and hints throughout the form.
Font Color	Determines the default color of the font used to render control text and hints throughout the form.
Font Weight	Determines the default style of the font used to render control text and hints throughout the form.
Text Align	Determines the justification of control text and hints throughout the form. Supported values are **left** and **right**.

Child Layout Tab

The setting in this tab apply to the labels of the root panes within the Form.

Fields on the Forms Child Layout Tab

Property	Description
Inherit from System Defaults	Check box determines whether or not the values on this tab are inherited from the system defaults.
Width	Determines the default width in pixels inherited by top-level panes. The width is that of the pane's child content area, excluding any space reserved for the pane label and hint.
Height	Determines the default height in pixels inherited by top-level panes. The height is that of the pane's child content area, excluding any space reserved for the pane label and hint.
Padding	Determines the default padding inherited by top-level panes. Padding is the spacing between adjacent sibling form elements. The value is a space-separated list of between one and four non-negative integers, representing the top, right, bottom and left padding respectively, in pixels. Missing values default to the last value in the list.

Property	Description
Margin	Determines the default margin inherited by top-level panes. Margin is the free space around the edges of a pane. The value is a space-separated list of between one and four non-negative integers, representing the top, right, bottom and left margins respectively, in pixels. Missing values default to the last value in the list.
BG Color	Determines the default background color inherited by top-level panes.
Border	Determines the default border style inherited by top-level panes. Supported values are **line** and **none**. A line-style pane border is drawn as a horizontal line beneath the pane label and only appears when the label position is **top**.
Overflow	Determines the default overflow strategy inherited by top-level panes. Overflow strategy determines how the pane responds to an explicit width and/or height setting that is less than the minimum required to display all of its content. Supported values are **expand**, **auto**, and **hidden**. The default strategy, **expand**, causes the pane to ignore a width or height setting if it is less than the minimum required. The **auto** strategy accepts the explicit size and displays scrollbars to enabled the hidden content to be revealed. The **hidden** strategy simply crops any content which lies outside the explicit bounds.

Child Labels Tab

Fields in the Forms Child Labels Tab

Property	Description
Inherit from System Defaults	Check box determines whether or not the values on this tab are inherited from the system defaults.
Label Width	Determines the default label width in pixels inherited by top-level panes.
Label Position	Determines the default label position inherited by top-level panes. Label position is with respect to the associated pane. Supported values are **left** and **top**.
Label Visible	Determines the default label visibility inherited by top-level panes.
Font Name	Determines the default label font inherited by top-level panes.
Font Size	Determines the default label font height (in points) inherited by top-level panes.
Font Color	Determines the default label text color inherited by top-level panes.
Font Weight	Determines the default label font style inherited by top-level panes. Supported styles are **normal** and **bold**.
Text Align	Determines the default label justification inherited by top-level panes. Supported values are **left** and **right**.

Rules Tab

Fields in the Forms Rules Tab

Property	Description
Event Type Open	Shows the rules to be triggered when the form is first opened.
Event Type Submit	Shows the rules to be triggered by the submit event of the form.
Event Type Close	Shows the rules to be triggered when the form is closed. This happens after the submit event, if there is one.
Event Type Localize	Shows the rules to be triggered when the form locale is changed.

Resources Tab

Fields on the Forms Resources Tab

Property	Description
Path	Displays a path to the resource.
Add (+) button	Adds a resource.
Delete (x) button	Removes the reference to the resource.

Preview Data Tab

Fields on the Preview Data Tab

Property	Description
Preview Data File	Select a file to furnish initial data values for the controls on the form. Choices are None, Default, or Custom. If **None** is selected, no values appear initially in the form controls. **Default** provides a default value for each type of control. To use **Custom**, first create a copy of the default `.data.json` file. Edit its values, and then select the file in the **Custom** field.
	It is also possible to create input data from the data submitted in preview. To do this, open the form in preview, fill out the values in the form, and click **Submit**. The submitted data is logged in the preview application. Copy the JSON object from the log, and paste it as the content of the custom `.data.json` file.

Properties View for Panes

The Properties view for a pane, whether it be a horizontal, vertical, tabbed, or message pane, contains nine tabs: General, Properties, Mappings, Layout, Font, Child Layout, Child Labels, Rules, and Mobile Properties. The Layout and Font tabs for panes are identical to those for controls.

General Tab

The Properties view for panes contain a General tab. This tab contains general properties for the pane currently selected in the canvas, and contains the fields shown in the table General Tab for Panes.

⚠️ When panes and controls are marked as disabled or invisible, the data normally displayed by these elements are still delivered to the browser. Therefore, making panes and controls disabled or invisible should not be used as a mechanism to protect sensitive data.

General Tab for Panes

Property	Description
Name	The name of the pane, used in JavaScript to refer to this object. The `Rename` button allows you to change the name using the `Enter the Name` dialog. The Name field only appears when the Solutions Design mode is active.
Label	The label for the pane that appears on the form (if the **Label Visibility** > **Visible** check box is selected). This property is bindable.
	See Setting Bindings and Setting Rules for more details.
Pane Type	A drop-down list showing the type of the pane. Allows you to select another type. If the object is a **Vertical Pane**, for instance, this setting can be used to change it to a **Horizontal Pane**.
Style Class Names	Specify a CSS class to be used for styling at the pane level.
Pane Data Type	Specifies the type of object that is set on the pane.
	If there is a value binding for a pane but the pane data type is not set explicitly, a warning message displays: `[pane_name]: pane mappings need an explicit pane data type`. You can set the explicit data type by using a quick fix, or by selecting the data type picker from the property sheet.
	For a pane data type determined by a binding, the interface also shows the data type.
Label Visibility	Determines whether the pane's label is visible. This value can either be inherited from the parent object of the pane, or set explicitly on the pane.
Visible	Determines whether the pane (together with its child elements) is visible. This value can be changed at runtime via scripting. This property is bindable.
	See Setting Bindings and Setting Rules for more details.
Enabled	Determines whether the controls within the pane can be modified or not. This value can be changed at runtime via scripting. This property is bindable.
	See Setting Bindings and Setting Rules for more details.

Property	Description
Read Only	Determines whether the pane (together with its child elements) is read-only. The check box is disabled for pane types, which do not support a read-only setting at runtime.

Properties Tab

The Properties tab contains special fields that pertain specifically to the type of pane being configured. Thus the Properties tabs on the Properties view for panes differ in their fields.

The horizontal pane, vertical pane, tabbed pane, and message pane have a common set of properties on the Properties tab. The grid pane and record pane have some additional properties.

The following sections describe the Properties tab for these panes separately.

Properties for Horizontal Pane, Vertical Pane, and Tabbed Pane

Property	Description
Static Rendering	Check box to enable static rendering for a pane. The information displayed within a static pane is displayed as read-only and you cannot modify the data. This property can be set only at design-time. It is not possible to convert a pane to static at runtime.
Text Only	Check box to render a static pane as plain text, without any control widgets.This property is enabled only if the **Static Rendering** property is selected.
Defer Rendering	Check box to defer the rendering of a pane until it is made visible. If the pane is visible at the time of loading, then it is rendered once the form is completely initialized and the Form Open event is fired. This property can be set only at design-time and it cannot be updated using bindings or using the API.
Defer Initialization	Check box to defer the initialization of the children of the pane until the pane is rendered. This means that the pane object itself is always instantiated and available, but any nested child is not initialized until the pane is about to be rendered. This property is enabled only if the **Defer Rendering** property is selected.

Properties for Message Pane

Property	Description
Suppress Validation Messages	Check box to suppress the display of messages from the modeled pane and control validations. The default value is `false`, in which case the message pane displays all messages, both modeled validations and those programmatically added using the API. If `true`, the message pane displays only programmatically added messages.

Record Pane Properties Tab

Property	Description
Support Add Operation (and Label)	Check box to render a button in the record pane that can add a new record to the end of the list being managed by the record pane. The default label is **New**, but can be overridden by providing a new value in the **Label** input box.
Support Delete Operation (and Label)	Check box to render a button in the record pane that can delete the currently viewed record. The default label is **Delete**, but can be overridden by providing a new value in the **Label** input box.
Show Navigator	Check box to display the navigation bar with the record pane, allowing navigation across the set of records in the record pane.
Selection	Click the **Add a Binding** button to specify the binding endpoint for a record pane. This can also be used when record pane is used in conjunction with a grid pane to offer a master/detail view of a list of objects. In such a scenario, the selection of the grid pane is bound to the selection of the record pane, and the value of the grid pane is bound to the value of the record pane. Whenever you select a different row in the grid pane, the corresponding record is shown in detail in the record pane.
Defer Rendering	Check box to defer the rendering of a pane until it is made visible. If the pane is visible at the time of loading, then it is rendered once the form is completely initialized and the Form Open event is fired. This property can be set only at design-time and it cannot be updated using bindings or using the API.
Defer Initialization	Check box to defer the initialization of the children of the pane until the pane needs to be rendered. This means that the pane object itself is always instantiated and available, but any nested child is not initialized until the pane is about to be rendered. This property is enabled only if the **Defer Rendering** property is selected.

Grid Pane Properties Tab

Property	Description
Visible Rows	Specify the maximum number of visible rows.
Support Add Operation	Check box to render a button in the record pane that can add a new record to the end of the list being managed by the record pane. The default label is **New**, but can be overridden by providing a new value in the **Label** input box.
Support Delete Operation	Check box to render a button in the record pane that can delete the currently viewed record. The default label is **Delete**, but can be overridden by providing a new value in the **Label** input box.
Buttons enabled in read only pane	Check box to render the buttons as enabled even if the grid pane is read-only.
Add/Delete Button Position	A radio control to define the position of the **Add** and **Delete** buttons. The supported values are **Top** and **Bottom**.

Property	Description
Movable Columns	Check box to enable movable columns. This feature is not supported in GWT runtime.
Sortable	Check box to enable sorting of the data in the grid pane.
Always render controls	Check box to render a grid pane such that the child controls are directly rendered in edit mode. It eliminates the additional click action required to activate edit mode of grid pane. This property is related to **Always Render** property for controls. Refer to Properties Tab for further details.
Static Rendering	Check box to enable static rendering for a pane. The information is displayed in a read-only mode within a static pane, and you cannot modify the data. This property can be set only at design-time. It is not possible to convert a pane to static at runtime.
Text Only	Check box to render a static pane as plain text, without any control widgets.This property is enabled only if the **Static Rendering** property is selected.
Defer Rendering	Check box to defer the rendering of a pane until it is made visible. If the pane is visible at the time of loading, then it is rendered once the form is completely initialized and the Form Open event is fired. This property can be set only at design-time and it cannot be updated using bindings or using the API.
Defer Initialization	Check box to defer the initialization of the children of the pane until the pane is rendered. This means that the pane object itself is always instantiated and available, but any nested child is not initialized until the pane is about to be rendered. This property is enabled only if the **Defer Rendering** property is selected.
Selection Model	Radio control used to specify the selection model. The supported values are **single** and **multiple**.
Selection	Selection of a binding endpoint for a grid pane or master-detail pane. Click the **Binding** button to open the Edit Binding dialog, which allows you to choose an item and specify the update behavior invoked for that item when an instance is selected in the grid pane or master pane.

Property	Description
Row Label	Used to specify the row label template resource and type. The available options are: • **External Reference**: to pick the row label from an external resource. You need to define the row labels in the `<row_labels>.properties` file. The resource key for the row label must follow the naming convention *`<component-name>[.property].<featureName>`*, and end with `.rowLabel` or `_rowLabel`. For example, `pane.grid.property.rowLabel=Attr1 {0}`. Resources that do not follow these conventions are not displayed in the Resource Picker. • **Custom**: to specify a user-defined row label By default, the value of the first column of the grid pane is used as the row label. This property is available only at accessible runtime.
Default Sort Column	Used to specify the column on which to sort by default.
Default Sort Order	Used to specify the default sort order - ascending or descending.

Modal Dialog Pane Properties Tab

Property	Description
Render **Close** button on the title bar	Check box to render a **Close** (**X**) button on the title bar of the modal dialog. Clicking this button fires the cancel event on the modal dialog.
Render **Close** button on the pane	Check box to render an explicit **Close** button on the modal dialog. The default label is **Close**, but you can change the label by providing a new value in the **Label** field. The **Close** button closes the modal dialog, and fires the close event. You need to provide the semantics of a close action, that is, make sure that any relevant changes, which occurred in the modal dialog, are propagated to the main form model.
Render **Cancel** button on the pane	Check box to render an explicit **Cancel** button on the modal dialog. The default label is **Cancel**, but you can change the label by providing a new value in the **Label** field. Note that a **Cancel** button closes the modal dialog, and fires the cancel event. You need to provide the semantics of a cancel action, that is, make sure that all the changes which occurred in the modal dialog are reverted.
Support **ESC** key to close	Check box to support the **Escape** key on the modal dialog. If enabled, the cancel event is fired when the user presses the **Escape** key.

Property	Description
Dialog position	Radio buttons to specify the position of the modal dialog. • **Center of the window**: Renders the dialog at the center of the window. • **Center of the form**: Renders the dialog at the center of the form irrespective of the scrolling position. • **Relative to the focused element**: Renders the dialog at the position specified relative to the focused element. The top left position of the element is treated as (0,0). You need to specify the X and Y coordinates accordingly. Positive values move the pane downward and to the right by specified pixels. • **Absolute position**: Renders the dialog at an absolute position irrespective of the scrolling. The top left point in the view port is treated as (0,0). You need to specify the X and Y coordinates accordingly. Positive values move the pane downward and to the right by specified pixels.

Mappings Tab

The Properties view for panes contain a Mappings tab. This tab is used to view, edit, and create mappings for the selected pane. You can refer to Mappings Tab section for further details.

Layout Tab

Same as for controls. See Layout Tab.

Font Tab

Same as for controls. See Font Tab.

Child Layout Tab

Fields in the Child Layout Tab

Property	Description
Inherit from System Defaults	Check box determines whether or not the values on this tab are inherited from the system defaults.
Width	Determines the width of child objects of this pane.
Height	Determines the height of child objects of this pane.
Padding	Sets the white-space gap between the outer edge of the child objects of this pane and their inner content. Specified as one to four implied pixel values applied in this order: top, right, bottom, and left. For example, 8 pixels of padding could be specified as **8**, or as four space-separated values: **8 8 8 8**.
Margin	Sets the gap between the border of the pane's child objects and their parent or sibling panes. Specified as one to four implied pixel values applied in this order: top, right, bottom, and left. For example, 4 pixels for margins could be specified as **4**, or as four space-separated values: **4 4 4 4**.

Property	Description
BG Color	Determines the background color of child objects of this pane.
Border	Sets a border around child objects of the pane. Possible values are **none** and **line**.
Overflow	Determines how child objects of the pane behave when their content exceeds their dimensions. Possible values are **expand**, **auto**, and **hidden**.

Child Labels

The settings in this property tab pertain to the child controls and panes of this pane. They have no effect on the label of the pane itself.

Fields in the Child Labels Tab

Property	Description
Inherit From Parent	Specifies whether the layout properties of the pane are inherited. If the **Inherit From Parent** check box is selected, all fields are disabled for editing. Clearing the **Inherit From Parent** field allows you to edit all fields on this tab.
Label Width	Determines the width of the label in pixels.
Label Position	Determines the label position inherited by child controls and panes. Label position is with respect to the associated control or pane. Supported values are **left** and **top**.
Label Visible	Determines the label visibility inherited by child controls and panes.
Font Name	Determines the label font face name inherited by child controls and panes.
Font Size	Determines the label font height (in points) inherited by child controls and panes.
Font Color	Determines the label text color inherited by child controls and panes.
Font Weight	Determines the label font style inherited by child controls and panes. Supported styles are **normal** and **bold**.
Text Align	Determines the label justification inherited by child controls and panes. Supported values are **left** and **right**.

Validations Tab

The Validations tab lists the validation scripts defined for the pane, and provides a mechanism to create new validation for that pane.

Fields in the Validation Tab

Fields	Description
Name	The name of the validation.

Fields	Description
Execute When	When the validation is executed. The options are: • On Form Submit • On Value Change
Script	The validation script.
Message Type	The type of validation message. The options are: • External Reference • Custom
Message	The validation message that is displayed in the message pane if your entry is invalid. This is either a static message defined in the validation, or a reference to a resource key, where the key begins with "validation_".
List	Check box used to specify whether the validation is to be executed on the complete list or for each value in the list for a multi-valued control. The functionality of the two states is as follows: true: The validation is invoked with the context.value set to the list value of a multi-valued control. false: The validation is invoked once for each value in the list, with context.value set to a specific value each time.
Enabled	Check box used to specify whether the validation is to be executed at runtime. The functionality of the two states is as follows: true: The validation is invoked at runtime. false: The validation is not invoked at runtime.
	This button opens the Define Validation dialog. The dialog contains two parts, a **Script** area for writing the validation script, and a Message area for typing the message that is displayed in a message pane if your entry is invalid. The Define Validation dialog allows you to specify when the validation script runs.
	This button deletes the selected validation.

Rules Tab

Similar to the Properties tab, not all panes have a Rules tab on their Properties view, and for those that do, the Rules tabs differ in their supported events.

The panes *with* a Rules tab include - Vertical Pane, Horizontal Pane, Record Pane, Grid Pane.

The panes *without* a Rules tab include - Tabbed Pane, Message Pane.

Fields in the Rules Tab

Property	Description
Event Type Double-click	Shows the rules to be triggered when a record in the pane is double-clicked.
Event Type Select	Shows the rules to be triggered when a record in the pane is selected.
Event Type Update	Shows the rules to be triggered when the value of the pane is updated.

For each pane, only the event types supported by that pane is listed in the tab.

Clicking the **Add Rule** button opens the New Rule wizard, with the corresponding event already added to the new Rule. To add a new rule, see Setting Rules.

Mobile Tab

Fields in the Mobile Tab

Property	Description
Short Label	Used to specify a short label which is displayed instead of the ordinary label for the mobile rendering of the form.
Toolbar	Used to mark one pane as the toolbar pane in a form which is targeted for mobile devices.

Properties View for Controls

The Properties view for a control contains eight tabs: General, Mappings, Properties, Layout, Font, Validations, Rules, and Mobile. The Layout and Font tabs for controls are identical to those for panes.

General Tab

The Properties view for controls contains a General tab. This tab contains general properties for the object currently selected in the Canvas

⚠️ When panes and controls are marked as disabled or invisible, the data normally displayed by these elements are still delivered to the browser. Therefore, making panes and controls disabled or invisible should not be used as a mechanism to protect sensitive data.

.

General Tab Fields

Property	Description
Name	Name of the control. Used in scripts to refer to the control. The `Rename` button allows you to change the name using the `Enter the Name` dialog. The Name field only appears when the Solutions Design mode is active.

Property	Description
Label	Text that appears next to the control. The value of the label can be bound to an input parameter so that the control can be dynamically labeled at runtime. Labels can be localized. This property is bindable. See Setting Bindings and Setting Rules for more details.
Control Type	A drop-down list showing the type of the control. Allows you to select another type. If the object is a `Date` control, for instance, this field can be used to change it to a `Time` or `DateTime` control.
Style Class Names	Specify a CSS class to be used for styling at the control level.
Hint	Text that provides a hint to help you complete the form correctly. For controls, the hint appears just beneath the control. Text for a hint can be mapped to the value of a parameter. Hints can be localized. This property is bindable. See Setting Bindings and Setting Rules for more details.
Value	At runtime, it is the value with which a control is initialized. **Value** is not supported for **Hyperlink** and **Image** controls. This property is bindable. See Setting Bindings and Setting Rules for more details.
Label Visibility	Whether the label for the control can be seen on the form. Values can be **Inherit**, **Visible**, or not visible (neither check box selected).
Visible	Determines whether the control is visible to you. This field can be bound to a parameter value, or its value can be set at runtime by an **Action** script, based on an event. This property is bindable. See Setting Bindings and Setting Rules for more details.
Enabled	Determines whether you can update the value of the control. This property is bindable. See Setting Bindings and Setting Rules for more details.
Required	Indicates whether you must provide a value for this control in order for the form to be successfully validated. At runtime, required fields are preceded by an asterisk, to indicate that the field is required. If you do not provide a value for a control that is required, the form cannot be submitted. This property is bindable. See Setting Bindings and Setting Rules for more details.
Tab Index	Determines the position of the element in the tabbing order for the form. The tabbing order determines the order in which elements on the form receive focus when the tab key is used to navigate from one element to another. This attribute is valid for all controls except Image and Label controls, where focus is irrelevant. See **Tabbing Navigation**, page 287 , for more details.

Tabbing Navigation

The **Tab Index** attribute can be used to determine the order in which elements receive focus as you navigate from field to field through a form with the tab key. The tabbing navigation behavior for a form is as follows:

1. Those elements on the form that support the Tab Index attribute, and assign a positive value to it, are navigated first. Navigation proceeds from the element with the lowest Tab Index value to the element with the highest value. Values need not be sequential, nor must they begin with any particular value. Elements that have identical Tab Index values are navigated in the order in which they appear on the form.

2. Those elements that do not support the Tab Index attribute, or support it and assign it a value of "0," are navigated next. These elements are navigated in the order in which they appear in the Outline view.

Mappings Tab

The Properties view for controls contain a Mappings tab. This tab is used to view, edit, and create mappings for the selected control. You can refer to Mappings Tab section for further details.

Properties Tab

The Properties tab contains special fields that pertain specifically to the type of control being configured. Thus, not all controls have a Properties tab on their Properties view, and for those that do, the Properties tabs differ in their fields.

The controls *with* a Properties tab include - Button, Date, Time, Date-Time, Hyperlink, Image, Optionlist, Passthrough, Radiogroup, Text, and Text Area.

The controls *without* a Properties tab include - Check box and Label.

The controls having an extra property on the Properties tab only if the control is a child of a grid pane include - Date, Time, Date-Time, Optionlist, Radiogroup, Text, and Text Area.

Property for Child Controls of Grid Pane

Property	Description
Always Render in Grid Pane	Check box to render the grid pane child controls directly in edit mode. This property is related to **Always render controls** property for grid pane. If the **Always render controls** property is set to `true`, then all the controls on a grid pane are directly rendered in edit mode. However, if the **Always render controls** property is set to `false`, then the **Always Render** property setting on each control determines whether or not the control is rendered in edit mode. Refer to Grid Pane Properties Tab for further details. This property is only supported in GWT runtime.

The details of the **Properties** tab for each control is described in the table Button Properties Tab.

Properties Tab for the Button Control:

Button Properties Tab

Property	Description
Button Type	Radiogroup list that allows the Form Designer to configure the type of button. Possible values are primary, peripheral, and associative. There are four kinds of buttons on the palette: **Button** (generic), **Cancel Button**, **Submit Button**, and **Close Button**.
	The Properties tab is identical for each of them, except that the default value in the **Button Type** list is **primary** for the Submit button, and **peripheral** for the other button types. This means that a Submit button that is placed on a form from the palette, by default, is invoked on a mouse click or when you press the **Enter** key. Buttons of the other types are invoked only when they are clicked or otherwise selected.

Properties Tab for the Date Control:

Date Control Properties Tab

Property	Description
Edit as List	Check box to enable the Date control to represent multiple date values. It enables you to add and delete items, or move them up and down.
Maximum Visible Rows	Specify the maximum number of visible rows.
Start Year	Specify the first year that should be displayed in the date picker in mobile forms. The default value is `-20`.
	This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
Start Year Relative	Check box used to specify whether the value of **Start Year** is interpreted as being relative to the current year or as an absolute year. The default value is `true`.
	This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
End Year	Specify the last year to be displayed in the date picker in mobile forms. The default value is `20`.
	This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
End Year Relative	Check box to specify whether the value of **End Year** is interpreted as being relative to the current year or as an absolute year. The default value is `true`.
	This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.

Properties Tab for the Time Control:

Time Control Properties Tab

Property	Description
Edit as List	Check box to enable the Time control to represent multiple time values. It enables you to add and delete items, or move them up and down.
Maximum Visible Rows	Specify the maximum number of visible rows.
Minute Increment	Specify the increment to be used while displaying the choice of minutes in a time control. The default value is 15 and the maximum value is 60. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.

Properties Tab for the Date-Time Control:

Date Control Properties Tab

Property	Description
Edit as List	Check box to enable the Date-Time control to represent multiple date-time values. It enables you to add and delete items, or move them up and down.
Maximum Visible Rows	Specify the maximum number of visible rows.
Start Year	Specify the first year that should be displayed in the date picker in mobile forms. The default value is -20. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for further details.
Start Year Relative	Check box used to specify whether the value of **Start Year** is interpreted as being relative to the current year or as an absolute year. The default value is true. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
End Year	Specify the last year to be displayed in the date picker in mobile forms. The default value is 20. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
End Year Relative	Check box to specify whether the value of **End Year** is interpreted as being relative to the current year or as an absolute year. The default value is true. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.
Minute Increment	Specify the increment to be used while displaying the choice of minutes in the date-time control. The default value is 15 and the maximum value is 60. This is a mobile forms specific configuration. See Mobile Specific Configuration of Controls and Panes for more details.

Properties Tab for the Hyperlink Control:

Hyperlink Properties Tab

Property	Description
URL	The URL for this control.
Link Text	The text for the hyperlink that appears on the form. This value can be set via script actions, computation actions, or bindings.

Properties Tab for the Image Control:

Image Properties Tab

Property	Description
URL	URL pointing to the image file that is to appear on the form.
	The URL can either be an absolute URL, or a special folder relative path to the form resource. If the path is relative, then the image resource to which it points are deployed automatically when the form resource is deployed.
	This value can be updated via script at runtime or by using a binding. If the location of the image is set dynamically to a relative path, then those resources are not be automatically deployed with the form. You can add these images as references in the form resources tab, so they are deployed when the form resource is deployed. See Configuration of Parameters for more details.

Properties Tab for the Optionlist Control:

Optionlist Properties Tab

Property	Description
Allow Multiple Selections	Allow users to choose multiple items from those listed, rather than being restricted to a single choice.
Choices: Binding	
Label Array	Use the **Add a Binding** button to: • Create a binding for this property • Update this property using a Computation Action
Value Array	Use the **Add a Binding** button to: • Create a binding for this property • Update this property using a Computation Action
Choices: External Reference	Click the ellipsis **(...)** button to choose an external object with value pairs, such as enumeration containing label values and name values.

Property	Description
Choices: Custom Values	Use this table to add (+), delete (x), or reorder the choices in this list.

Properties Tab for the Pass-through Control:

Pass-through Control Properties Tab

Property	Description
Markup	Used to specify a block of HTML fragment. This markup is inserted directly into the browser DOM at runtime. Click the **Binding** button to set the markup via a binding or computation action.

Properties Tab for the Radiogroup Control:

Radiogroup Control Properties Tab

Property	Description
Format	Choose the format for this control: auto, columns, horizontal, or vertical
Columns	Choose number of columns to display the radio buttons: 1, 2, or more
Choice Layout	
Layout type	Select one of the following: • Auto • Columns • Horizontal • Vertical
Columns	Select number of columns.
Choices: Bindings	
Label Array	Use the **Add a Binding** button to: • Create a binding for this property • Update this property using a Computation Action
Value Array	Use the **Add a Binding** button to: • Create a binding for this property • Update this property using a Computation Action
Choices: External Reference	

Property	Description
Select object	Click the ellipsis **(...)** button to choose an object, such as an Enumeration from a business object model, that contains name-value or label-value pairs.
Choice: Custom Values	
Manage the List	Use this table to add (+), delete (x), or reorder the choices that are part of this list.

Properties Tab for the Text Control:

Text Properties Tab

Property	Description
Edit as List	Check box to enable the Text control to represent multiple text values. It enables you to add and delete items, or move them up and down.
Maximum Visible Rows	Specify the maximum number of visible rows.
Secret	A control that visually masks what is input in order to prevent eavesdropping. Typically used when you type a password.
Numeric	A control with this option selected treats the contents of the text field as a number with respect to how the decimal point is localized. This allows the control to work in locales that use a different symbol (such as ",") for the decimal point.
Format	The **Format** options are enabled only if the **Numeric** property is selected. The supported values are **External Reference** and **Custom**. See Numeric Controls for more information.
Maximum Length	Maximum length of the text field, in numbers of characters.
Display Length	The length of the field that can be viewed at one time, in numbers of characters.

Properties Tab for the Text Area Control:

Text Area Properties Tab

Property	Description
Edit as List	Check box to enable the Text Area control to represent multiple text values. It enables you to add and delete items, or move them up and down.
Maximum Visible Rows	Specify the maximum number of visible rows.
Rows	Determines the number of lines that can be typed in the textarea control.

Property	Description
Columns	Determines the number of characters that can be typed in a single line of the textarea control.
Maximum Length	Maximum length of the text area, in numbers of characters.

The controls with an extra property on the Properties tab only if the control is a child of a grid pane include - Date, Time, Date-Time, Optionlist, Radiogroup, Text, and Text Area.

Property for Child Controls of Grid Pane

Property	Description
Always Render	Check box to render the grid pane child controls directly in edit mode. This property is linked to **Always render controls** property of grid pane. If the **Always render controls** property is set to `true`, then all the controls on a grid pane are directly rendered in edit mode. However, if the **Always render controls** property is set to `false`, then the **Always Render** property setting on each control determines whether or not the control is rendered in edit mode. Refer to Grid Pane Properties Tab for further details. This property is only supported in GWT runtime.

Layout Tab

All Properties views for controls contain a Layout tab, and all Layout tabs contain the same fields. The fields listed in the table Layout Tab appear on the Layout tab for forms and for all panes and controls.

Layout Tab

Property	Description
Inherit From Parent	Specifies whether the layout properties of the control are inherited. If the **Inherit** check box is selected, all fields are disabled for editing. Clearing the **Inherit** field allows you to edit all fields on this tab.
Width	Width of the pane or control. The width is that of the *content area*. For panes, this is the area occupied by child panes and controls; for controls, it is the area occupied by the control body, excluding label and hint areas.
Height	Height of the pane or control. The height is that of the *content area*. For panes, this is the area occupied by child panes and controls; for controls, it is the area occupied by the control body, excluding label and hint areas.
BG Color	Background color for the object being configured.
Padding	Sets the white-space gap between the outer edge of the object and its inner content. Specified as one to four implied pixel values applied in this order: top, right, bottom, and left. For example, 8 pixels of padding could be specified as **8**, or as four space-separated values: **8 8 8 8**.

Property	Description
Margin	Sets the gap between the object's border and its parent or sibling objects. Specified as one to four implied pixel values applied in this order: top, right, bottom, and left. For example, 4 pixels for margins could be specified as **4**, or as four space-separated values: **4 4 4 4**.
Border	Sets a border around the object. Possible values are **none** and **line**.
Overflow	Determines how the control or pane behaves when its content exceeds its dimensions. Possible values are expand, auto, and hidden. These terms are described here: • **expand** The pane expands to show all of its contents. (Manual values for a pane or control's width or height that are less than the preferred width or height are ignored when the overflow mode is `expand`.) • **auto** The pane uses scroll bars to show any content that cannot fit within the fixed width and height. • **hidden** Any content that exceeds the prescribed width and height is not be shown.

Font Tab

Font Tab for Controls

Property	Description
Inherit from Parent	If selected, the font settings are inherited from the parent pane. This check box is selected by default for all controls and panes. Top level panes inherit their font settings from the form itself. At the form level, the equivalent default setting is **Inherit from System Defaults**. Clearing one of these **Inherit** check boxes makes the remaining fields on the Font tab editable.
Font Name	A selection of standard browser-supported font names.
Font Size	The size of the font. Values can be chosen from the drop-down list or typed in.
Font Color	The color of the font, chosen from a standard color picker.
Font Weight	The weight of the font. Possible values are `normal` and `bold`.
Text Align	Alignment of text. Possible values are `left` and `right`.

Validations Tab

Validations Tab for Controls

Property	Description
Name	The name of the validation.

Property	Description
Execute When	When the validation is executed. The options are: • On Form Submit • On Value Change
Script	The validation script.
Message Type	The type of validation message. The options are: • External Reference • Custom
Message	The error message that is displayed in the message pane if your entry is invalid.
List	Check box used to specify whether the validation is to be executed on the complete list or for each value in the list for a multi-valued control. The functionality of the two states is as follows: `true` : The validation is invoked when the `context.value` is set to the list value for a multi-valued control. `false` : The validation is executed once for each value in the list, with `context.value` set to a specific value each time.
	This button opens the Define Validation dialog. The dialog contains two parts, a **Script** area for writing the validation script, and a Message area for typing the message that is displayed in a message pane if your entry is invalid. The Define Validation dialog allows you to specify when the validation script is run.
	This button deletes the selected validation.

Rules Tab

The Rules tab lists the Rules triggered by each of the events supported by the Control, and provides a mechanism to create new Rules for that control.

Fields in the Controls Rules tab

Property	Description
Event Type Enter	Shows the rules to be triggered when the control gains focus.
Event Type Exit	Shows the rules to be triggered when the control looses focus.
Event Type Update	Shows the rules to be triggered when the value of the control changes.
Event Type Select	Shows the rules to be triggered when the control is selected, such as when a button is clicked.

For each Control, only the event types supported by that control is listed in the tab.

Clicking the **Add Rule** button opens the New Rule wizard, with the corresponding event already added to the new Rule. To add a new rule, see Setting Rules.

Mobile Tab

The Mobile tab is used for mobile specific configuration.

Property	Description
Short Label	Specify a short label which is displayed instead of the **Label** for the mobile rendering of the form.

Configuration of Parameters

To configure a parameter, you need to define a few fields, such as Name, Label, Type, and so on.

Define the following fields:

- **Name**

  This field is only seen if the Solution Design capability is enabled. The Rename button shows a rename dialog.

- **Label**

  Business name of the parameter.

- **Mode In**

  The value is treated as read-only.

- **Mode Out**

  There is no value provided at form load, but the form may provide a value during submit.

- **Mode In/Out**

  The value may be read and written.

- **Type**

  One of the following:

  - **Text**

    Supporting single-line and multiple-line strings

  - **Integer**

    Supporting 32-bit integers

  - **Decimal**

    Supporting 64-bit double precision floating point numbers

  - **Boolean**

    .

  - **Date**

    Supporting localized display

  - **Time**

Supporting localized display

– **DateTime**

Supporting localized display. Precision to number of seconds.

- **Length**

Used only for Text, Integer and Decimal types

- **Decimal Places**

Used only for the Decimal type

- **Bindings**

Shows bindings and computation actions involving this parameter.

Context Menus

Context menus are available in the Outline view as well as in the Form Designer canvas.

Outline View Context Menu

You can use a context menu from the Outline view.

For more details, see Usage of the Outline View with Forms.

Form Designer Canvas Context Menu

You can also use a context menu from the canvas, by right-clicking the form icon or any form element in the Outline view. On the context menu, options are displayed as per the selected element.

Manage Form Elements from the Outline View

Select	Definition
Cut (Ctrl+X)	Available for all elements except for fixed categories (Data, Shared Actions, Rules)
Copy (Ctrl+C)	Available for all elements. After you copy an element to the clipboard, you can paste it within this form or another form.
Paste (Ctrl+V)	Available when content is available on clipboard
Delete	Available for all elements except for fixed nodes (Data, Shared Actions, Rules) and for the form itself
Rename	Available for all elements except for fixed categories (Data, Shared Actions, Rules), as well as for added actions and rules
Select All (Ctrl+A)	Selects all root panes. Select All does not select parameters, shared actions, or rules.
Show Properties view	Shows the Properties view, if not previously visible.

Keyboard Shortcuts

The use of keyboard shortcuts increases efficiency. Many keyboard shortcuts are available for all types of forms, including the ones rendered in accessible mode.

When a form is rendered, initially the focus is on the first component of the form.

Generic Keyboard Shortcuts

Press	To Do
Tab	Shifts the focus to the next component in the form.
Shift+Tab	Shifts the focus back to the previous component in the form.

Grid Panes

This section summarizes the keyboard shortcuts you can use for grid panes.

Grid Panes in Display Mode

Grid panes can operate either in display mode or in edit mode. The edit widget does not pop up in display mode when the focus is on the cell. When the focus first shifts to a grid pane, the pane is in display mode.

The keyboard shortcuts listed in the table Keyboard Shortcuts for Grid Panes in Display Mode are applicable only to display mode.

Keyboard Shortcuts for Grid Panes in Display Mode

Press	To Do
Enter, or F2, or Click	Activates edit mode, and selects the row. The focus is set on the control in the current cell. For non-editable grids, clicking or pressing Enter selects the row in a single-select grid, or toggles the row selection in a multi-select grid.
Delete	Deletes the selected row. The focus is set on the same cell of the next row.
Tab	Shifts the focus to the navigation bar if the grid is paginated. If the grid is not paginated and the command bar is visible, the focus shifts to the command bar. If both, the navigation bar and the command bar are not visible, the focus shifts to the next component in the form.
Shift+Tab	Shifts the focus to the last column heading of the grid pane.
Up Arrow key	Shifts the focus to the same cell in the previous row. If the focus is on the first visible row of the table, and the paginated grid pane has a previous page, the focus shifts to the same cell in the last row of the previous page. If the focus is already on the first visible row of the first page, it remains on the same cell.
Down Arrow key	Shifts the focus to the same cell in the next row. If the focus is on the last visible row of the table, and the paginated grid pane has a next page, the focus shifts to the same cell in the first row of the next page. If the focus is already on the last visible row of the last page, it remains on the same cell.

Press	To Do
Left Arrow key	Shifts the focus to the previous focusable cell in the same row. If none of the previous cells in the same row is focusable, the focus shifts to the last focusable cell in the previous row.
Right Arrow key	Shifts the focus to the next focusable cell in the same row. If none of the next cells in the same row is focusable, the focus shifts to the first focusable cell in the next row.
Page Up key	Displays the previous page of rows when the grid pane is paginated. The focus stays on the same cell on the displayed page of records.
Page Down key	Displays the next page of rows when the grid pane is paginated. The focus stays on the same cell on the displayed page of records.
Home key	Shifts the focus to the first column of the first visible row.
End key	Shifts the focus to the first column of the last visible row.
Ctrl+Home	Shifts the focus to the first column of the first row in the entire record set.
Ctrl+End	Shifts the focus to the first column of the last row in the entire record set.

Grid Panes in Edit Mode

When in edit mode, the controls in each cell are displayed as editable when the cell has the focus.

If a control is disabled or read-only, then it continues to display the text version of the control value. The grid pane does not handle any of these keys if the active cell editor handles the keystroke. For example, the `textarea` controls handle Up/Down Arrow keys. Pressing these keys affects the `textarea` and not the grid pane.

The keyboard shortcuts listed in the table Keyboard Shortcuts for Grid Panes in Edit Mode are applicable only to edit mode.

Keyboard Shortcuts for Grid Panes in Edit Mode

Press	To Do
Enter, or Escape, or Ctrl +Enter	Activates display mode, and updates the value. Validations run, and the focus remains on the recently edited cell, which is now in display mode. The Enter key within a `textarea` or list control is not handled by the grid pane. For such cases, use Ctrl+Enter to activate display mode.

Press	To Do
Tab	Shifts the focus to the next cell. The grid pane remains in edit mode, and the editor for the next cell is activated. If the focus is currently on a cell in the last column, the focus shifts to one of the following elements (in this order): • the navigation bar if it is visible • **Add** or **Delete** buttons if they are enabled • the next component on the form If the focus is on the last cell of the last visible row, the focus shifts to the grid navigation bar, or the next component in the form if the grid pane is not paginated.
Shift+Tab	Shifts the focus to the previous cell. The grid pane remains in edit mode, and the editor for the previous cell is enabled. If the focus is in the first cell of the first visible row, it shifts to the grid column headers.
Up Arrow key, or Ctrl+Up Arrow key	Shifts the focus to the cell in the same column in the previous row. Grid panes do not handle the Up Arrow key within a few controls, such as Textarea, Optionlist, Radiogroup, or List control. For such cases, use Ctrl+Up Arrow key.
Down Arrow key	Shifts the focus to the cell in the same column in the next row. Grid panes do not handle the Up Arrow key within a few controls, such as Textarea, Optionlist, Radiogroup, or List control. For such cases, use Ctrl+Down Arrow key.
Page Up key	Displays the previous page of rows. The focus shifts to the upper-left cell of the new page of records.
Page Down key	Displays the next page of rows. The focus shifts to the upper-left cell of the new page of records.

Grid Pane Column Headers

The heading for each column is rendered as an HTML anchor tag. As they are rendered as hyperlinks, each column heading is a tab stop when traversing the form.

For sortable grids, pressing the Enter key activates the hyperlink, and sorts the rows on that column between three possible states: unsorted (the default), sorted ascending, and sorted descending.

The keyboard shortcuts listed in the table Keyboard Shortcuts for Grid Pane Column Headers are applicable only to the grid pane column headers.

Keyboard Shortcuts for Grid Pane Column Headers

Press	To Do
Tab	Shifts the focus to the next heading in the grid pane header row. If the focus is on the heading of the last column, it shifts the focus to the first cell in the grid pane content.

Press	To Do
Shift+Tab	Shifts the focus to the previous heading in the grid pane header row. If the focus is on the heading of the first column, it shifts to the previous component in the form.
Enter	Changes the sorting state of the column to the next state. The states are ascending, descending, or unsorted (original).

Grid Pane Navigation Bar

For grid panes with more rows than a single page can accommodate, a navigation bar appears at the bottom of the grid pane. By using it, you can go to the first, previous, next, or last page of the pane.

If you press Tab when the focus is on the navigation bar, the focus shifts to the "First" or the "Next" arrows. The "First" and "Previous" arrows are disabled when the first page of the grid pane is displayed. Similarly, the "Next" and the "Last" arrows are disabled when the last page is displayed.

The keyboard shortcuts listed in the table Keyboard Shortcuts for Grid Pane Navigation Bar are applicable only to the grid pane navigation bar.

Keyboard Shortcuts for Grid Pane Navigation Bar

Press	To Do
Tab	Shifts the focus either to the next arrow on the navigation bar, or to the **New** and **Delete** buttons if they are enabled. If there is no arrow or button available, the focus shifts to the next component in the form after the grid pane.
Shift+Tab	Shifts the focus to the previous arrow on the navigation bar. If there is no arrow available, the focus shifts to the last cell in the last row of the grid pane.
Left Arrow, and Right Arrow keys	Shift the focus within the arrows on the navigation bar.
Enter	Invokes the currently focused arrow on the navigation bar.

List Controls

This section summarizes the keyboard shortcuts you can use for list controls.

List Controls in Display Mode

Similar to grid panes, list controls are either in display mode or in edit mode. If you press Tab when the focus is on a list control, the control is rendered in display mode, and the focus shifts to the first item in the list.

The keyboard shortcuts listed in the table Keyboard Shortcuts for List Controls in Display Mode are applicable only to list controls in display mode.

Keyboard Shortcuts for List Controls in Display Mode

Press	To Do
Enter, or Click	Activates edit mode, and maintains the focus on the current value.

Press	To Do
Delete	Deletes the selected item in the list. The focus shifts to the next item in the list, or to the **Add** button.
Tab	Shifts the focus to the list control command bar.
Shift+Tab	Shifts the focus to the previous component in the form.
Up Arrow key	Shifts the focus to the previous item in the list.
Down Arrow key	Shifts the focus to the next item in the list.
Home	Shifts the focus to the first item in the list.
End	Shifts focus to the last item in the list.

List Controls in Edit Mode

The keyboard shortcuts listed in the table Keyboard Shortcuts for List Controls in Edit Mode are applicable only to list controls in edit mode.

Keyboard Shortcuts for List Controls in Edit Mode

Press	To Do
Enter, or Escape	Activates display mode, and maintains the focus on the current value.
Ctrl+Enter	Activates display mode when editing a text area in the list.
Tab	Shifts the focus to the next value in the list. If the focus is already on the last value, it shifts to the list control command bar.
Shift+Tab	Shifts the focus to the previous value in the list. If the focus is already on the first value in the list, it shifts to the previous component in the form.
Up Arrow key	Shifts the focus to the previous value in the list. If the focus is already on the first value, it remains on that value.
Down Arrow key	Shifts the focus to the next value in the list. If the focus is already on the last value, it remains on that value.

List Control Command Bar

The keyboard shortcuts listed in the table Keyboard Shortcuts for List Control Command Bar are applicable only to the list control command bar.

Keyboard Shortcuts for List Control Command Bar

Press	To Do
Tab	Shifts the focus to the next component in the form after the list control.
Shift+Tab	Shifts the focus back to the content of the list control.
Left Arrow, and Right Arrow keys	Shift the focus within the control buttons (that is, add, delete, up, and down) in the list control command bar.
Enter	Invokes the currently focused control button.

Record Panes

This section summarizes the keyboard shortcuts you can use for record panes.

The keyboard navigation is just the same within a record pane. There are a few more keyboard shortcuts listed in the next sub-sections.

Record Pane Body

The keyboard shortcuts listed in the table Keyboard Shortcuts for Record Pane Body are applicable only to the record pane body.

Keyboard Shortcuts for Record Pane Body

Press	To Do
Page Up key	Displays the previous record in the list without shifting the focus. If the displayed record is the first one, there is no change.
Page Down key	Displays the next record in the list without shifting the focus. If the displayed record is the last one, there is no change.
Tab	Shifts the focus to the next control within the record pane.
Shift+Tab	If pressed when the first control has the focus, it shifts the focus to the central text field in the navigation bar, which displays the current record number.

Record Pane Navigation Bar

If you press Tab when focus is on the component before a record pane, the focus shifts to the central text field in the navigation bar, which displays the current record number.

The keyboard shortcuts listed in the table Keyboard Shortcuts for Record Pane Navigation Bar are applicable only to the record pane navigation bar.

Keyboard Shortcuts for Record Pane Navigation Bar

Press	To Do
Tab	Shifts the focus to the first component in the record pane.

Press	To Do
Shift+Tab	Shifts the focus to the previous component in the form.
Left Arrow, and Right Arrow keys	Shift the focus within the control buttons (that is, first, previous, current, next, and last) in the navigation bar.
Enter	Invokes the currently focused control button.

Tabbed Panes

This section summarizes the keyboard shortcuts you can use for tabbed panes.

Keyboard Shortcuts for Tabbed Panes

Press	To Do
Tab	If you press Tab when the focus is on the component before a Tabbed Pane, the focus shifts on the currently active tab. If you press Tab when the focus is on a tab in the tab bar, the focus shifts to the first control in the body of the currently active tab pane.
Shift+Tab	Shifts the focus back to the previous component in the form.
Left Arrow / Right Arrow	Shift the focus within the tabs in the tabbed pane.
Space	Makes the currently focused tab active.

CSS Classes

TIBCO Business Studio Forms supports the use of Cascading Style Sheets (CSS) for customizing how a form is rendered. This approach provides more flexibility and opportunities for reuse of style information than manually setting properties at the form model level.

This section lists the built-in CSS classes you can use. For general information on how to use CSS in TIBCO Business Studio Forms, see Cascading Style Sheets.

Built-in Static CSS Classes

When a form is rendered, there are a set of built-in CSS classes that are used at the Form, Pane, and Control level. You can use these CSS classes in defining custom rendering for these types of objects. The classes shown in this table are always rendered in the HTML DOM.

Built-in Static CSS Classes

CSS Class	Description
TibcoForms	Applied at the root node of the form, modal dialog pane, and the edit popup dialog in a grid pane.
tf-uir-enabled	When the enhanced user interface is enabled, applied at the root node of the form, modal dialog pane, and the edit popup dialog in a grid pane.

CSS Class	Description
tf-uir-disabled	When the enhanced user interface is disabled, applied at the root node of the form, modal dialog pane, and the edit popup dialog in a grid pane.
pane	Applied at the root node of each pane.
pane-vertical	Applied at the root node of each vertical pane, along with the pane class.
pane-horizontal	Applied at the root node of each horizontal pane, along with the pane class.
pane-tabbed	Applied at the root node of each tabbed pane, along with the pane class.
pane-grid	Applied at the root node of each grid pane, along with the pane class.
pane-grid-content	Applied to the underlying HTML table that contains the header row and values of a grid pane.
pane-grid-content-header-row	Applied to the row in the grid pane that contains column headers.
pane-grid-sortable	Applied to the header row of a grid pane whose columns are sortable.
pane-grid-sort-asc	Applied to the header label of a column that is currently sorted in ascending order.
pane-grid-sort-desc	Applied to the header label of a column that is currently sorted in descending order.
pane-grid-content-odd-row	Applied to odd rows in a grid pane
pane-grid-content-even-row	Applied to even rows in a grid pane
pane-messages	Applied at the root node of each messages pane, along with the pane class.
pane-record	Applied at the root node of each record pane, along with the pane class.
pane-label	Applied at the node that contains the label of a pane. This is nested within the node that has the pane class set.
pane-content	Applied at a node that contains all the child controls and panes of the parent pane.
pane-content-left	When applied at the pane level, left-aligns the contents of a pane.

CSS Class	Description
component	Applied at the root node of each control or pane. So each node that has a class **pane-content** contains 0 or more nodes with a class **component**.
label	Applied at a node within a **component** that contains the label for the control or pane.
container	**Deprecated.** Use **tf-container** instead of **container**. Applied at a node within a **component** that contains the content of the control or pane.
tf-container	Applied at a node within a **component**. Contains the content of the control or pane.
control-widget	Applied on the specific element used for the control, such as an `<input>` element for text controls. This is a descendent of the node that contains the **tf-container** class.
hint	Applied to the node that contains a hint for a control. This is a descendent of the node that contains the **tf-container** class.
control-textinput	Applied at the same node as the **component** class for text controls.
control-textarea	Applied at the same node as the **component** class for textarea controls.
control-date	Applied at the same node as the **component** class for date controls.
control-time	Applied at the same node as the **component** class for time controls.
control-datetime	Applied at the same node as the **component** class for datetime controls.
control-checkbox	Applied at the same node as the **component** class for checkbox controls.
control-optionlist	Applied at the same node as the **component** class for optionlist controls.
control-radiogroup	Applied at the same node as the **component** class for radiogroup controls.
control-image	Applied at the same node as the **component** class for image controls.
control-label	Applied at the same node as the **component** class for label controls.

CSS Class	Description
control-hyperlink	Applied at the same node as the **component** class for hyperlink controls.
control-duration	Applied at the same node as the **component** class for duration controls.

Built-in Dynamic CSS Classes

A set of CSS classes are used to define when controls and panes are in certain states such as `required` and `disabled`. All of these classes are added to the same level as the component class when needed.

Built-in Dynamic CSS Classes

CSS Class	Description
required	Added when the control is required.
disabled	Added when the control or pane is disabled.
invalid	Added when the control has failed validation.
tf-form-loaded	Added to the `iframe` element when the form is loaded in an `iframe`, and indicates that a form is currently displayed in the `iframe`. The class selector `tf-form-not-loaded` is removed from the `iframe` element.
tf-form-not-loaded	Added to the `iframe` element if an `iframe` is used to load the form, but currently the form is not displayed in the `iframe`. The class selector `tf-form-loaded` is removed from the `iframe` element.
[custom]	Custom classes defined in the form designer or set dynamically via the `setClass()` API are added at the same level as the **component** class.

Common Resource Keys

This section lists all the resource keys that are provided as a part of the common resources bundle. The keys are grouped into their basic functional areas, and the default values in the base bundle are given for reference.

For the details on how to override the default values or add new resource keys to the bundle, see Property Resource Bundles.

Keys for Number Patterns

This section lists the resource keys for formatting values in number controls.

The number control shows resource keys that begin with "`format_`". You can override their values, and also add new keys that begin with "`format_`".

For these resource keys, the number grouping separator is always represented as the comma meta-character, and the decimal separator is always represented as the period meta-character. The actual grouping and separator characters are translated separately, exactly once. It is not necessary to translate these grouping and separator meta-characters at every place where they appear.

For more information on how to specify a number format, see Numeric Controls.

Number Patterns

Resource Key	Reference Value	Description
format_currency	\u00A4#,##0.00; (\u00A4#,##0.00)	Specifies a basic currency format. The unicode character \u00A4 represents a currency symbol, which is substituted at runtime. For example: *$123,344.89* for positive numbers, *($34,121.00)* for negative numbers
format_integer	#,##0	Basic grouped integer format. For example: *123,456*
format_integer_ungrouped	0	Basic ungrouped integer format. For example: *123456*
format_decimal	#,##0.###	Basic decimal format. For example: *123,456.123*
format_decimal_1	#,##0.0	Decimal format showing exactly one decimal place. For example: *123.1*
format_decimal_2	#,##0.00	Decimal format showing exactly 2 decimal places. For example: *123.10*
format_decimal_3	#,##0.000	Decimal format showing exactly 3 decimal places. For example: *123.100*
format_decimal_4	#,##0.0000	Decimal format showing exactly 4 decimal places. For example: *123.1000*
format_decimal_ungrouped	0.###	Basic ungrouped decimal format. For example: *123456.123*

Keys for Basic Number and Currency Symbols

This section lists the resource keys for values that get substituted in the numeric formats.

For example, number_grouping substitutes the "," character in the numeric formats.

Basic Number and Currency Symbols

Resource Key	Reference Value	Description
number_decimal	.	The decimal point that is substituted for the "." meta-character.
number_grouping	,	The grouping separator that is substituted for the "," meta-character.
number_zero	0	The character to be used as the leading zero in numeric formats.
currency_symbol	$	Must be translated only for specific countries. The currency symbol that is substituted for \u00A4.
currency_decimal	.	Used when the currency format is used.
currency_grouping	,	Used when the currency format is used.
currency_code	USD	The standard 3-letter currency code.

Keys for Duration Control Labels

This section lists the resource keys for the labels of duration controls.

For example, duration_label_years substitutes "Years" in the text input field.

Resource Key	Reference Value	Description
duration_label_years	Years	Labels the text input field as "Years".
duration_label_months	Months	Labels the text input field as "Months".
duration_label_days	Days	Labels the text input field as "Days".
duration_label_hours	Hours	Labels the text input field as "Hours".
duration_label_minutes	Minutes	Labels the text input field as "Minutes".
duration_label_seconds	Seconds	Labels the text input field as "Seconds".

Resource Key	Reference Value	Description
duration_label_milliseconds	Milliseconds	Labels the text input field as "Milliseconds".
format_duration_years	{0} years	Used for the text representation of duration, where {0} is greater than 1. For example: "2 years"
format_duration_months	{0} months	Used for the text representation of duration, where {0} is greater than 1
format_duration_days	{0} days	Used for the text representation of duration, where {0} is greater than 1.
format_duration_hours	{0} hours	Used for the text representation of duration, where {0} is greater than 1.
format_duration_minutes	{0} minutes	Used for the text representation of duration, where {0} is greater than 1.
format_duration_seconds	{0} seconds	Used for the text representation of duration, where {0} is greater than 1.
format_duration_milliseconds	{0} milliseconds	Used for the text representation of duration, where {0} is greater than 1.
format_duration_years_singular	{0} year	Used for the text representation of duration, where {0} is equal to 1.
format_duration_months_singular	{0} month	Used for the text representation of duration, where {0} is equal to 1
format_duration_days_singular	{0} day	Used for the text representation of duration, where {0} is equal to 1.
format_duration_hours_singular	{0} hour	Used for the text representation of duration, where {0} is equal to 1.
format_duration_minutes_singular	{0} minute	Used for the text representation of duration, where {0} is equal to 1.

Resource Key	Reference Value	Description
format_duration_seconds_singular	{0} second	Used for the text representation of duration, where {0} is equal to 1.
format_duration_ milliseconds_singular	{0} millisecond	Used for the text representation of duration, where {0} is equal to 1.
duration_separator	,	Separates the values in the text representation of duration.
duration_order	yMdHmsS	The order in which the specific duration units appear in the text representation of duration, where y = years, M = months, d = days, H = hours, m = minutes, s = seconds, S = milliseconds.
duration_items_display_sep	\ /\	Separates the values in the text representation of list items. Note: All the other list controls use "items_display_sep" as defined in List Control Keys . This new separator is necessary to separate duration items in a list, because the "duration_separator" that formats a duration value also uses a "," in the base bundle.

Duration Control Labels

Keys for Date-Time Patterns

This section lists the resource keys for date-time controls.

Supported Date Time Keys

Resource Key	Reference Value	Description
date_month_abbrev	['Jan','Feb','Mar','Apr ','May','Jun','Jul','Aug ','Sep','Oct','Nov','De c']	Used for the short names of the months in a calendar control. Specified as a JavaScript array.
date_month	['January','February', 'March','April','May', 'June','July','August',' September','October' ,'November','Decem ber']	Used for the long names of the months in a calendar control. Specified as a JavaScript array.

Resource Key	Reference Value	Description
date_day_abbrev	['Sun','Mon','Tue','Wed','Thu','Fri','Sat']	Used for the short names of the days of the week in a calendar control. Always begins with Sunday. Specified as a JavaScript array.
time_ampm	['AM','PM']	Used for the Latin abbreviations for the 12-hour clock convention.
datetime_date_label	Date	Labels the date portion of a date-time control.
datetime_time_label	Time	Labels the time portion of a date-time control.
accessible_date_label	{0} (enter as {1})	Accessible Forms: Used to augment the label for date, time, and date-time controls. {0} is substituted with the original control label, and {1} is substituted with the edit format used for the control.
date_today	Today	Used in the Date Picker. Clicking this label takes the date control to today's date.
date_format	MMM dd, yyyy	Used to display date values in a date-time control. This is a standard Java date format string.
date_time_format	MMM dd, yyyy hh:mm:ss a	Used to display date and time values in a date-time control. This is a standard Java date format string.
time_format	hh:mm:ss a	Used to display values in a time control. This is a standard Java date format string.
date_edit_format	MM/dd/yyyy	Used when users are expected to edit a date value directly in the text box. The format must be kept simple. You can modify the sequence of the year, month, and day; and then change the separators.
date_time_edit_format	MM/dd/yyyy HH:mm:ssZ	Used when users are expected to edit a date-time value directly in the text box. The format must be kept simple. You can modify the sequence of the year, month, day, hours, minutes, and seconds; and then change the separators.

Resource Key	Reference Value	Description
time_edit_format	HH:mm:ssZ	Used when users are expected to edit a time value directly in the text box. The format must be kept simple. You can modify the sequence of the hours, minutes, and seconds; and then change the separators.
date_picker_ok_label	OK	Labels the OK button in the time and date-time control pickers.
date_first_day_of_week	0	Used to indicate the first day of the week when displaying a calendar. If 0 is specified, the first day of a week is Sunday. If 1 is specified, the first day of a week is Monday, and so on.
date_hours_circle_basis	24	Used by the Date Picker to show the hours in 24-hour or 12-hour clock.

Unsupported Date Time Keys

Resource Key	Reference Value	Description
date_month_narrow	['J','F','M','A','M','J','J','A','S','O','N','D']	Used for narrow, one letter abbreviations of the months in a calendar control. Specified as a JavaScript array.
date_day	['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday']	Used for the long names of the days of the week in a calendar control. Always begins with Sunday. Specified as a JavaScript array.
date_day_narrow	['S','M','T','W','T','F','S']	Used for the narrow, one-letter abbreviations of the days of the week in a calendar control. Always begins with Sunday. Specified as a JavaScript array.
date_era_long	['Before Christ','Anno Domini']	Used for the complete text of era names in formatted dates. For example: Before Christ
date_era	['BC','AD']	Used for the short forms of era names in formatted dates. For example: BC
time_24hour	true	Used to determine whether the time is to be displayed in a 24-hour clock format.

Keys for Optionlist Controls

This section lists the resource key for drop-down list controls.

Optionlist Key

Resource Key	Reference Value	Description
option_select_label	- Select -	The value initially displayed in a drop-down list before the user makes a selection.

Keys for Built-in Buttons

This section lists the resource keys for built-in buttons.

Resource Key	Reference Value	Description
form_cancel_label	Cancel	Labels the **Cancel** button that is generated by default or is added from the palette.
form_submit_label	Submit	Labels the **Submit** button that is generated by default or is added from the palette.
form_close_label	Close	Labels the **Close** button that is generated by default or is added from the palette.
spinner_done_label	Done	Mobile Forms: Indicates that the user has picked a value.
spinner_cancel_label	Cancel	Mobile Forms: Indicates that the user has cancelled the operation of picking a value.
screen_back_label	Back	Mobile Forms: Returns to the previously viewed screen.
screen_add_list_item_label	+	Mobile Forms: Label on the button to add a new value or a new record. Must be a single character.

Keys for Grid and Record Panes

This section lists the resource keys for grid panes and record panes.

Grid and Record Pane Keys

Resource Key	Reference Value	Description
pane_new_label	New	Used as the default label for adding a new record to a collection pane representing a composition reference.
		You can override it on specific instances of grid or record panes.

Resource Key	Reference Value	Description
pane_delete_label	Delete	Used as the default label for deleting an existing record from a collection pane representing a composition reference. You can override it on specific instances of grid or record panes.
pane_add_label	Add	Used as the default label for adding a reference to an existing object to a collection pane representing a non-aggregation reference. You can override it on specific instances of grid or record panes.
pane_remove_label	Remove	Used as the default label for removing a reference to an existing object from a collection pane representing a non-aggregation reference. You can override it on specific instances of grid or record panes.
msgd_pane_confirm_delete_label	Delete {0} selected records?	Used as a confirmation message when users delete multiple records from a multi-select grid pane.
ssgd_pane_confirm_delete_label	Delete the selected record?	Used as a confirmation message when users delete a single record from a grid pane.
grid_pane_page_info	\ {0} - {1} of {2} \	Gives pagination information of the grid pane navigation bar. It shows the number of active records and the total number of records. For example: *11-20 of 35*
rp_confirm_delete_label	Delete the current record?	Used as a confirmation message when users delete the displayed record from a record pane.

Resource Key	Reference Value	Description
nav_first_label	First	Used as the hover help for the record and grid pane control button that navigates users to the first page of records in a paginated grid pane, or to the first record in a record pane.
nav_last_label	Last	Used as the hover help for the record and grid pane control button that navigates users to the last page of records in a paginated grid pane, or to the last record in a record pane.
nav_next_label	Next	Used as the hover help for the record and grid pane control button that navigates users to the next page of records in a paginated grid pane, or to the next record in a record pane.
nav_previous_label	Previous	Used as the hover help for the record and grid pane control button that navigates users to the previous page of records in a paginated grid pane, or to the previous record in a record pane.
record_record_label	Record	Used only in the record pane navigation panel. Used in combination with record_record_of_label to display "*Record x of y*" on the User Interface, where *x* is a drop-down list showing the current record, and *y* is the total number of records.
record_record_of_label	of	Used only in the record pane navigation panel. Used in combination with record_record_label to display "*Record x of y*" on the User Interface, where *x* is a drop-down list showing the current record, and *y* is the total number of records.
record_pane_record_info	Record {0} of {1}	Used only in the record pane navigation panel. Appears in the title of the record number field in the navigation panel.

Resource Key	Reference Value	Description
`accessible_gd_pane_select_row_label`	Select row to edit or delete	Accessible Forms: The label for radiogroup/checkbox of the grid pane selection cell in accessible runtime. Rendered as offscreen text.
`accessible_gd_pane_select_all_rows_label`	Select all rows	Accessible Forms: The label for radiogroup/checkbox of the multi-select grid pane selection header in accessible runtime. Rendered as offscreen text.
`accessible_gd_pane_normal_col_header_label`	Click to sort in ascending order	Accessible Forms: The label used in the header of a grid pane in accessible runtime when sorting is not in effect.
`accessible_gd_pane_asc_ord_col_header_label`	Sorted in ascending order. Click to sort in descending order.	Accessible Forms: The label used in the header of a grid pane in accessible runtime when the column is sorted in ascending order.
`accessible_gd_pane_desc_ord_col_header_label`	Sorted in descending order. Click to remove sorting.	Accessible Forms: The label used in the header of a grid pane in accessible runtime when the column is sorted in descending order.

Keys for Modal Dialog Panes

This section lists the resource keys for modal dialog panes.

Modal Dialog Pane Keys

Resource Key	Reference Value	Description
`pane_close_label`	OK	Used as the default label for the explicit **Close** button that closes the dialog. You can override it on specific instances of modal dialog panes.
`pane_cancel_label`	Cancel	Used as the default label for the **Cancel** button that cancels the modal dialog. You can override it on specific instances of modal dialog panes.

Resource Key	Reference Value	Description
`dialog_pane_close_button_tooltip`	Close	Used as the tooltip message for the **Close** (**X**) button on the title bar of a modal dialog pane. You can override it on specific instances of a modal dialog pane.

Keys for Built-in Validation Messages

This section lists the resource keys for built-in validation messages.

Built-in Validation Message Keys

Resource Key	Reference Value	Description
`form_validation_error_message`	Error in script for validation {0} of Control {1} ({2})\: {3}	Used to display a message for a script error while running a validation. {0} is the name of the validation. {1} is the name of the control. {2} and {3} are debugging messages.
`form_action_error_message`	Error in script for action {0} ({1})\: {2}	Used to display a message for a script error while running an action. {0} is the name of the action. {1} and {2} are debugging messages.
`form_required_message`	{0} is a required field.	Used to display a message when a required value is missing. {0} is the label of the control.
`record_pane_error_label`	There are errors on record(s) {0}.	Mobile Forms: Used to display a message for errors on multiple records. {0} is a comma separated list of numbers.
`nested_pane_error_label`	There are errors on this screen.	Mobile Forms: Used to display a message for validation failures on one or more components on the current pane.

Keys for List Controls

This section lists the resource keys for list controls.

List Control Keys

Resource Key	Reference Value	Description
list_add_label	add	Used as the hover help for the **Add** button in list controls.
list_delete_label	delete	Used as the hover help for the **Delete** button in list controls.
list_move_up_label	up	Used as the hover help for the **Up** button in list controls.
list_move_down_label	down	Used as the hover help for the **Down** button in list controls.
items_display_sep	,	Used as a separator when displaying text representation of items in a list.
static_items_display_sep	|	Used as a separator when displaying text representation of items in a list, where the items already use the basic separator. For example: *1 year, 2 months | 2 years, 5 months*

Keys for Implicit Validation Messages

This section lists the resource keys for implicit validation messages.

These messages are used when validations are automatically generated based on the underlying BOM specification of the value. In all these messages, the value {0} is substituted with the label of the control that fails the validation.

Implicit Validation Messages

Resource Key	Reference Value	Description
validation_date_format	"{0}" is incompatible with ISO format 'yyyy-MM-dd'	Used when the target value must be a proper ISO 8601 formatted date. See http://www.w3.org/TR/NOTE-datetime .
validation_time_format	"{0}" is incompatible with ISO format 'HH:mm:ssZ'	Used when the target value must be a proper ISO 8601 formatted time. See http://www.w3.org/TR/NOTE-datetime .

Resource Key	Reference Value	Description
validation_datetime_format	"{0}" is incompatible with ISO format 'yyyy-MM-dd'T'HH:mm:ssZ'	Used when the target value must be a proper ISO 8601 date-time value. See http://www.w3.org/TR/NOTE-datetime .
validation_decimal_fixed_point	"{0}" must be a fixed point decimal number with no more than {1} digits and {2} decimal places	Used for BOM attributes and process data fields that are configured as fixed point decimal numbers in the Resources tab of the BOM editor.
validation_decimal_floating_point	"{0}" must be a floating point decimal number	Used for BOM attributes that are configured as floating point decimal numbers in the Resources tab of the BOM editor.
validation_integer_length	"{0}" must be an integer with no more than {1} digits	Used to specify a length constraint on the number of digits of Integer type BOM attributes and process data fields.
validation_integer	"{0}" must be an integer.	Used for BOM attributes and process data fields of the Integer type.
validation_text_length	"{0}" must be a value with no more than {1} characters	Used for BOM attributes of the Text type that have a length constraint.
validation_lower_limit_inclusive	"{0}" must be a number greater than or equal to {1}	Used for numbers that have a lower limit specified (including the limit value).
validation_lower_limit	"{0}" must be a number greater than {1}	Used for numbers that have a lower limit specified (excluding the limit value).
validation_upper_limit_inclusive	"{0}" must be a number less than or equal to {1}	Used for numbers that have an upper limit specified (including the limit value).
validation_upper_limit	"{0}" must be a number less than {1}	Used for numbers that have an upper limit specified (excluding the limit value).
validation_multiplicity_maximum	"{0}" must contain at most {1} values	Used when an upper limit is specified for child multiplicity. For example: *0..5*

Resource Key	Reference Value	Description
`validation_multiplicity_exact`	"{0}" must contain exactly {1} values	Used when an exact number is specified for child multiplicity, and the number is greater than 1. For example: *3*
`validation_multiplicity_minimum`	"{0}" must contain at least {1} value(s)	Used when only lower limit is specified for child multiplicity, without an upper limit. For example: *1..** or *3..**
`validation_multiplicity_range`	"{0}" must contain between {1} and {2} values	Used when an exact multiplicity range is specified with both a lower and an upper limit. Both the numbers must be non-zero and the upper limit must be greater than the lower limit. For example: *1..5* or *2..4*
`validation_numeric`	"{0}" must be a number	Used for BOM attributes and process data fields of the Integer or Decimal type.
`validation_pattern`	"{0}" has the wrong format for the "{1}" data type	Used for BOM attributes of the Text type that specify a regular expression constraint pattern.
`validation_null_global_data_ref`	Error in server data: The value for the parameter "{0}" contains an invalid global data object reference	Used when the initial data sent from the server is invalid and cannot be corrected within the form. This is typically reported when there are references to global data that have been deleted. Users can still submit or close the form. The respective parameter value cannot be modified in the form. The bound components are disabled and any attempt to update them using the API causes an error.

Keys for Enhanced User Interface

This section lists the common resource keys associated with the enhanced user interface.

For more information, see Enhanced User Interface.

Resource Keys for Enhanced User Interface

Resource Key	Reference Value	Description
form_class_defs		Unused in the default implementation. Defines the classes to be applied at the form-level.
form_perf_ui_defs	[{"selector": "", "classes": "modal-content"}, \ {"selector": "div/table/tbody/tr/td/div/div/div/table", "classes": "table table-condensed"}, \ {"selector": "div/table/tbody/tr/td/div/div/div/table/tbody/tr/td/div/div", "classes": "nav-tabs-label"}, \ {"selector": "div/table/tbody/tr/td/div/div/div/div", "classes": "panel panel-default"}, \ {"selector": "div/table/tbody/tr/td/div/div/div/div/div/table", "classes": "table table-striped table-bordered table-condensed"}, \ {"selector": "div/table/tbody/tr/td/div/div/div/div/table", "classes": "table table-striped table-bordered table-condensed"}, \ {"selector": "div/table/tbody/tr/td[2]", "classes": "modal-header"}, \ {"selector": "div/table/tbody/tr/td[2]/div/table/tbody/tr/td/label", "classes": "label label-visible"}]	Defines the set of classes to be applied for performance metrics user interface.

Resource Key	Reference Value	Description
`control_class_defs`	[{"selector": "div/div[2]", "classes": "help-block"}, \ {"selector": "div/div[1]", "classes": "form-group has-feedback"}, \ {"selector": "div/div[1]", "where": "in-grid", "classes": "form-group-sm"}, \ {"selector": "div/div/ span[tf_uir=feedback]", "classes": "glyphicon form-control-feedback"}, \ {"selector": "div/div[1]", "when": "invalid", "classes": "has-error"}, \ {"selector": "div/div/ span[tf_uir=feedback]", "when": "invalid", "classes": "glyphicon-warning-sign"}, \ {"selector": "div/div[1]", "when":"valid", "classes": "has-success"}, \ {"selector" : "div/div/ span[tf_uir=feedback]", "when": "valid", "classes": "glyphicon-ok"}]	Defines the set of classes to be applied at each element of a control markup.

Resource Key	Reference Value	Description
control_textinput_class_defs	[{"selector": "//input", "target": "widget", "classes": "form-control"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]", "classes": "input-group"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]/span[1]", "where": "not-in-grid", "classes": "input-group-addon"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]/span[1]/span[1]", "where": "not-in-grid", "classes": "fa fa-slack"}]	Defines the set of classes to be applied at each element of a text control markup.
control_textarea_class_defs	[{"selector": "//textarea", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a textarea control markup.
control_checkbox_class_defs	[{"selector": "//div/div[class=tf-container]/div[1]", "where": "not-in-grid", "classes": "input-group"}, \ {"selector": "//span", "target": "widget", "classes": "form-control"}, \ {"selector": "//div/div[class=tf-container]/div[1]/span[1]", "where": "not-in-grid", "classes": "input-group-addon"}, \ {"selector": "//div/div[class=tf-container]/div[1]/span[1]/i", "where": "not-in-grid", "classes": "glyphicon glyphicon-ok"}]	Defines the set of classes to be applied at each element of a checkbox markup.

Resource Key	Reference Value	Description
control_duration_class_defs	[{"selector": "//div/div/ div[1]", "target": "widget", "classes": "help-block"}, \ {"selector": "//div/div/ input", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a duration markup.
control_duration_class_defs	[{"selector": "//div", "target": "widget", "classes": "form-control tf-label-control"}]	Defines the set of classes to be applied at each element of a label markup.
control_optionlist_class_defs	[{"selector": "//select", "target": "widget", "classes": "form-control control-lg tf-select-fixer"}, \ {"selector": "div[class=tf-container]/div[1]", "classes": "input-group"}]	Defines the set of classes to be applied at each element of a single-select optionlist markup.
control_radiogroup_class_defs	[{"selector": "//div/div/div/span", "target": "widget", "classes": "radio-inline"}]	Defines the set of classes to be applied at each element of a radiogroup markup.
control_hyperlink_class_defs	[{"selector": "//a", "target": "widget", "classes": "btn-link"}, \ {"selector": "//div/div[2]/ div[1]", "classes": "has-feedback has-tibco-feedback"}]	Defines the set of classes to be applied at each element of a hyperlink markup.

Resource Key	Reference Value	Description
`control_button_class_defs`	[{"selector": "//div[tf-comp-type=button primary]/div[2]/div[1]/button", "classes": "btn btn-primary"}, \ {"selector": "//div[tf-comp-type=button peripheral]/div[2]/div[1]/button", "classes": "btn btn-secondary"}, \ {"selector": "//div[tf-comp-type=button associative]/div[2]/div[1]/button", "classes": "btn btn-link"}]	Defines the set of classes to be applied at each element of a button markup.
`control_date_class_defs`	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date control markup.
`control_datetime_class_defs`	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date-time control markup.
`control_time_class_defs`	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a time control markup.
`pane_vertical_class_defs`	[{"selector": "", "classes": "panel panel-default"}, \ {"selector": "div[class=pane-label]", "classes": "panel-heading"}, \ {"selector": "div[class=tf-container]", "classes": "panel-body"}]	Defines the set of classes to be applied at each element of a vertical pane markup.

Resource Key	Reference Value	Description
pane_horizontal_class_defs	[{"selector": "", "classes": "panel-default form-inline"}, \ {"selector": "div[class=pane-label]", "classes": "panel-heading"}, \ {"selector": "div[class=tf-container]", "classes": "panel-body"}]	Defines the set of classes to be applied at each element of a horizontal pane markup.
pane_grid_class_defs	[{"selector": "", "classes": "panel panel-default form-inline"}, \ {"selector": "div[class=pane-label]", "classes": "panel-heading"}, \ {"selector": "div[class=tf-container]", "classes": "panel-body"}, \ {"selector": "div[class=tf-container]/div", "classes": "form-horizontal"}, \ {"selector": "div[class=tf-container]/div/table", "classes": "table table-striped grid-line-table"}, \ {"selector": "div[class=tf-container]/div/button", "classes": "btn btn-secondary"}]	Defines the set of classes to be applied at each element of a grid pane markup.

Resource Key	Reference Value	Description
pane_record_class_defs	[{"selector": "", "classes": "panel panel-default"}, \ {"selector": "div[class=pane-label]", "classes": "panel-heading"}, \ {"selector": "div[class=tf-container]", "classes": "panel-body"}, \ {"selector": "div[class=tf-container]/ div[class=pane-page-navigation]/ div[class=nav-input]/ input", "classes": "form-control"}, \ {"selector": "div[class=tf-container]/div/button", "classes": "btn btn-secondary"}]	Defines the set of classes to be applied at each element of a record pane markup.
pane_tabbed_class_defs	[{"selector": "", "classes": "panel panel-default"}, \ {"selector": "div[class=pane-label]", "classes": "panel-heading"}, \ {"selector": "div[class=tf-container]", "classes": "panel-body"}, \ {"selector": "div[class=tf-container]/table/ tbody/tr/td/div/ div[class=gwt-Label]", "classes": "nav-tabs-label"}]	Defines the set of classes to be applied at each element of a tabbed pane markup.

Resource Key	Reference Value	Description
pane_modaldialog_class_defs	[{"selector": "//div", "classes": "modal-content"}, \ {"selector": "div/table/tbody/tr[1]/td[2]", "classes": "modal-header"}, \ {"selector": "div/table/tbody/tr[2]/td[2]/div/div/div[2]/button[class=tf-dialog-ok]", "classes": "btn btn-primary"}, \ {"selector": "div/table/tbody/tr[2]/td[2]/div/div/div[2]/button[class=tf-dialog-cancel]", "classes": "btn btn-secondary"}]	Defines the set of classes to be applied at each element of a modal dialog pane markup.
pane_messages_class_defs	[{"selector": "div/div", "classes": "pane-message-validation-error"}]	Defines the set of classes to be applied at each element of a message pane markup.
control_static_class_defs	[{"selector": "div[class=tf-container]/div[class=hint]", "classes": "help-block"}]	Defines the set of classes to be applied at each element of a static control markup.

Resource Key	Reference Value	Description
control_textinput_static_class_defs	[{"selector": "//input", "target":"widget", "classes": "form-control"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]", "where": "not-in-grid", "classes": "input-group"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]/span[1]", "where": "not-in-grid", "classes": "input-group-addon"}, \ {"selector": "//div[tf-comp-type=textinput numeric]/div[class=tf-container]/div[1]/span[1]/span[1]", "where": "not-in-grid", "classes": "fa fa-slack"}]	Defines the set of classes to be applied at each element of a text control markup in a static pane.
control_date_static_class_defs	[{"selector": "//input", "target":"widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date control markup in a static pane.
control_datetime_static_class_defs	[{"selector": "//input", "target":"widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date-time control markup in a static pane.
control_time_static_class_defs	[{"selector": "//input", "target":"widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a time control markup in a static pane.
control_duration_static_class_defs	[{"selector": "//input", "target":"widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a duration control markup in a static pane.

Resource Key	Reference Value	Description
`control_optionlist_static_class_defs`	[{"selector": "//input", "target":"widget", "classes": "form-control"}, \ {"selector": "//select", "target":"widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of an optionlist markup in a static pane.
`control_radiogroup_static_class_defs`	[{"selector": "div[class=tf-container]/span", "classes": "radio-inline"}]	Defines the set of classes to be applied at each element of a radiogroup markup in a static pane.
`control_textarea_static_class_defs`	[{"selector": "//textarea", "target": "widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a textarea markup in a static pane.
`control_label_static_class_defs`	[{"selector": "//div", "target": "widget", "classes": "form-control tf-label-control"}]	Defines the set of classes to be applied at each element of a label markup in a static pane.
`control_hyperlink_static_class_defs`	[{"selector": "div[class=tf-container]/div[1]/span", "classes": "glyphicon glyphicon-link"}, \ {"selector": "//a", "target": "widget", "classes": "btn-link"}]	Defines the set of classes to be applied at each element of a hyperlink markup in a static pane.
`control_button_static_class_defs`	[{"selector": "//button", "target": "widget", "classes": "btn-secondary"}, \ {"selector": "div", "where": "not-in-grid", "classes": "help-block"}]	Defines the set of classes to be applied at each element of a button markup in a static pane.
`pane_vertical_static_class_defs`	[{"selector": "", "classes": "panel panel-default"}, \ {"selector": "div[1]", "classes": "panel-heading"}, {"selector": "div[2]", "classes": "panel-body"}]	Defines the set of classes to be applied at each element of a vertical pane markup in a static pane.

Resource Key	Reference Value	Description
pane_horizontal_static_class_defs	[{"selector": "", "classes": "panel-default form-inline"}, \ {"selector": "div[1]", "classes": "panel-heading"}, {"selector": "div[2]", "classes": "panel-body"}]	Defines the set of classes to be applied at each element of a horizontal pane markup in a static pane.
control_list_class_defs	[{"selector": "div[2]", "target": "widget", "classes": "btn btn-tertiary"}, \ {"selector": "div[2]/span/a", "target": "widget", "classes": "btn-xs"}, \ {"selector": "div[2]/span[1]/a/span", "target": "widget", "classes": "glyphicon glyphicon-plus"}, \ {"selector": "div[2]/span[2]/a/span", "target": "widget", "classes": "glyphicon glyphicon-minus"}, \ {"selector": "div[2]/span[3]/a/span", "target": "widget", "classes": "glyphicon glyphicon-chevron-up"}, \ {"selector": "div[2]/span[4]/a/span", "target": "widget", "classes": "glyphicon glyphicon-chevron-down"}]	Defines the set of classes to be applied at each element of a list control markup.
control_textinput_list_class_defs	[{"selector": "//input", "target": "list-edit-widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a text list control markup.
control_date_list_class_defs	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "list-edit-widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date list control markup.

Resource Key	Reference Value	Description
`control_datetime_list_class_defs`	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "list-edit-widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a date-time list control markup.
`control_duration_list_class_defs`	[{"selector": "//div/div/div", "target": "list-edit-widget", "classes": "help-block"}, \ {"selector": "//div/div/input", "target": "list-edit-widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a duration list control markup.
`control_time_list_class_defs`	[{"selector": "//div/table/tbody/tr[1]/td[1]/input", "target": "list-edit-widget", "classes": "form-control"}]	Defines the set of classes to be applied at each element of a time list control markup.
`control_textarea_list_class_defs`	[{"selector":"//textarea", "target":"list-edit-widget", "classes":"form-control"}]	Defines the set of classes to be applied at each element of a textarea control markup.
`control_custom_class_defs`		Optional key. It's not defined by default, but you can use it to write class definitions similar to the ones that are provided for built-in controls.

Miscellaneous Keys

This section lists miscellaneous resource keys.

Miscellaneous Resource Keys

Resource Key	Reference Value	Description
`data_preview_empty`	There is no data to display.	Used as a data preview message for empty data.

Resource Key	Reference Value	Description
forms_compact_mode	[1], [2], [3], [1,2], [1,2,3], [2,3], [1,3], or empty to disable the key	The key applies to all controls and panes on a form to make the form smaller in size. When the value contains 1, the width of the grid panes in the form is set to a maximum of 600 pixels. When the value contains 2, the labels align to the top even if the child labels are configured to be aligned to the left. However, if the pane has only controls in it, the labels are not aligned to the top. When the value contains 3, it reduces the spacing between controls along with the spacing between labels and value fields within a control. The default value for the run time is [3]. If you want to disable the compact mode, specify an empty value for it in the custom property resource bundle.
align_toolbar_left	true, false	Unused by default. Aligns the contents of the toolbar pane to the left when set to true. The toolbar buttons, such as **Submit**, **Close**, and **Cancel** are aligned to the right by default. For large forms that need horizontal scrolling, you can align the buttons to the left using this key.

Design-time Constraints

You can configure the rules that the Validation Builder applies to all form models.

The Validation Builder applies the following categories of rules:

- Core
- General
- Resources
- JavaScript
- Forms Synchronization
- Components
- GWT/Mobile

You can change the configuration of these issues from the Errors/Warnings page in the Form Designer on the Preferences dialog. For more information, see Form Builders and Form Validation.

Client-side Validations

At runtime, a component is validated depending on how you configure them at design time - on value change, or on form submission.

To understand how the validations occur, see the Table: Runtime Constraints.

Runtime Constraints

Runtime Constraints	On Value Change	On Form Close	On Form Submit
BOM Constraint	✅	✅	✅
User-defined (**On Value Change**)	✅	✅	✅
User-defined (**On Form Submit**)	❌	❌	✅
Required	❌	❌	✅

If you configure a validation on form submission, it occurs when the user submits the form, or when the `validate(true)` API is called on the component, or parent pane, or the form.

If a validation configured on form submission fails for a component, the runtime invokes all the validations of that component on its value change until all the validations pass again. In such a case, it does not consider if the validation is configured on form submission or on value change. The validation messages displayed for controls as the result of a failed form submission disappear after the user provides a valid value.

Scripting

You can enhance the functionality of your forms by writing JavaScript code snippets on certain tabs in the Properties view.

There are two contexts where scripts may be added:

- **Actions**

  Actions may contain script, and are invoked as a part of one or more rules in response to a triggering event.

 To learn more, see Actions and Setting Rules.

- **Validations**

  Validations are scripts that determine you have specified a valid value for a control. When you specify a validation script, you configure it to run either when the form is submitted, or when the value for the control changes.

  To learn more, see Form Builders and Form Validation and Data Validation in a Form.

Business Analysis Capability versus Solution Design Capability

To create or modify scripts as shown in this section, you must ensure that the Solution Design capability is enabled. You can change modes by clicking the **Capability** button on the TIBCO Business Studio toolbar to open the drop-down list, if you are not already in the desired mode:
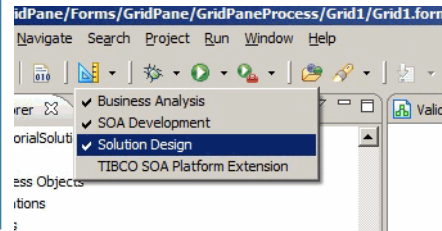
Forms Scripting Scope of Variables

These tables cover the various places with the form model that scripting is allowed, and describe the default script variables that are in scope in those places.

When setting a control value via the "f" array, the changes are not realized until the whole action script ends. This means that any bindings or rules that are tied to the updating of that control is not triggered until the whole script finishes. Use the setValue() method for the control whose value you are modifying.

The "f" array and "p" array functionality is deprecated. You can use control.*<control-name>*.getValue() instead of using "f" array and use p.get*<parameter-name>* instead of using the "p" array. See the table Action for details.

Action

Variable	Description
context	**read-only.** This is a data structure that provides access to the context under which the action is invoked. There are 6 fields available within this variable: • **context.control:** The control object that was the source of the event that triggered the rule. If the source was not a control, then this field is `null`. • **context.form:** The form object where the event originated. • **context.oldValue:** Provides the old value if this is a control or parameter update event. • **context.newValue:** Provides the new value if this is a control or parameter update event. • **context.pane** : The pane object that was the source of the event that triggered the rule. If the source was not a pane, then this field is `null`. • **context.record:** This field is provided within the computation actions where the destination control or pane is under a collection pane (grid or record pane). The record corresponds to the object in the destination control's (or pane's) parent pane value. For example, when you are computing the value of a control at the 6th row of a grid pane, the record points to the complex object at index 5 of the grid pane value. This field can also be used within validations.
control	Use `control.<control-name>` to access any control defined within the form.
data	Use `data.get<param-name>` to access the values of form parameters or data fields. This method returns either a primitive value for simple types such as Text and Boolean, or instances of objects when the type is defined in a BOM. For primitive types, `data.set<param-name>` is also available.
factory	Use `factory.<package-name>` to access factories based on packages defined within the business object models available to the form. These factories allow you to create new instances of classes defined in that package.
pane	Use `pane.<pane-name>` to access any pane defined within the form.
pkg	Use `pkg.<package-name>` to access package objects based on packages defined within the business object models available to the form. The package object allows you access definitions of Enumerations defined within the package.

Variable	Description
resource	Use `resource.<external-resource-name>.<property-name>` to access the localized values from property files. A property file can be added to the Presentation Resources folder and it can be referenced from a form by creating an External Resource in the form. For example: when a property file in the Presentation Resources folder is added as an External Resource in the form with the name `resource1`, all the properties in that file can be accessed in a user-defined form action script from the object returned by `resource.resource1`. Thus if the property file contains a property with name `name1`, the value of this property can be retrieved in a user-defined script as: `resource.resource1.name1`. If a localized bundle is provided and a value exists for the property in that bundle, the value from that bundle is returned. If the property is missing in the localized bundle, the value from the base bundle is returned. Use `resource.mappings.<parameter-name>` to access the values in the data mappings file of the form. These keys represent the properties to which all the controls or panes in the form are bound. The Form Designer automatically generates the data mappings file in the Presentation Resources folder, and it can be referenced from a form as an External Resource. For more information, see Data Mappings File .
f	**read-only**. Field value array that accesses the current values of controls in the form. Field values can be accessed using `f.controlName`. Field values can be updated by assigning a new value to them. Example: `f.foo='newValue';` Deprecated. Use `control.cn.setValue(cv)` instead of `f.cn = cv;` and `var cv = control.cn.getValue();` instead of `var cv = f.cn;`
p	**read-only**. Parameter value array that accesses the inbound values of parameters. `p` can replace a pane and control. Parameter values can be accessed using `p.paramName`. Deprecated. Use `data.setPn(pv);` instead of `p.pn = pv;` and `var pv = data.getPn();` instead of `var pv = p.pn;`
this	**read-write**. For actions that are initiated from a control event, `this` refers to the control object from which the event is initiated. From `this`, access the `form` object and other controls and make updates to the state of the form model. Deprecated. Use the new `context` variable that is available within the script.

Validation

Variable	Description
f	**read-only**. Field value array that accesses the current values of controls in the form. Field values can be accessed using `f.<control-name>`. Deprecated. Use `var cv = control.cn.getValue();` instead of `var cv = f.cn;`.

Variable	Description
`this`	**read-only**. Refers to the control object upon which the validation is configured. From `this`, access the form object and other controls, although no updates to the form model are allowed within a form validation script. Deprecated. Use `context.value` to get the value of the control being validated.
`context`	**read-only.** Within the scope of a validation script, the context variable supports only the following field: `context.value`: This is equal to the value of the control being validated. If the control is multi-valued, such as a text control with the list setting enabled, then the validation is run once for each value in the list.
`resource`	User-defined validation scripts can retrieve localized values from property files using the `resource` variable. The `resource` details provided in the table Action are also applicable for validation scripts.

Forms Scripting Order of Script Execution

The scripts specified in the form model are executed in a certain order during the different form life-cycle events.

Errors in user-provided scripts are caught and logged at the error level at runtime and shown in the preview page logging area in the Form Designer.

The order of script execution is as follows:

When the Form is opened

- **Form Open** event is published.

When the Submit button is clicked

- Control validation scripts are executed.
- **Form Submit** event is published.
- **Form Close** event is published.

When the Close button is clicked

- **Form Close** event is published.

When the Cancel button is clicked

- **Form Cancel** event is published.
- **Form Close** event is published.

When You Update a Control Value and Navigate out of the Control

- Control Validation scripts are executed.
- **Control Exit** event is published.
- **Control Update** event is published.

For the events where the validation scripts are executed, no further steps proceed if any of the validations fail.

API for Scripting

You can access the form model at runtime by using API methods in your JavaScript scripts.

The API described here can be used for writing *validations* as well as *actions*. Updating form fields or their properties using **set** methods is allowed only in the context of actions. Validation scripts cannot modify the fields or their properties.

Methods for Form Class

The table lists the methods for the Form class.

Form Class

Method	Return Value	Description
getClassName()	String	Returns custom CSS classnames set on the form. The value may be null, a single CSS classname, or a space-separated list of CSS classnames that are applied to the root level of the form.
getControl(String controlName)	Control	Returns the control with the given name.
getLocale()	String	Returns the string representation of the locale currently being used to render the form.
getPane(String paneName)	Pane	Returns the pane with the given name.
getPanes()	Pane[]	Returns an array of root panes of this form.
getParameterValue(String paramName)	Object List Array	Returns the value of the parameter or data field with the given name. This is either a list, an array, or a duration object, BOM JavaScript wrapper object or native JavaScript Boolean, Date, Number or String object, depending on the type of parameter.
invokeAction(String actionName, Object control), Context context		Invokes the shared or default action specified by the actionName parameter. The object passed as the control parameter is used as the this variable inside the script of the invoked action. The third argument is used as the context variable in the invoked script. If the third argument is null, then a default context is used in the invoked script. Example usage: `context.form.invokeAction('submit', this, context);` Either a shared action defined for the form or one of the pre-defined actions can be used with the invokeAction method. The pre-defined actions are: submit, apply, close, cancel, validate, and reset.

Method	Return Value	Description
isNumber(Object value, Integer length)	Boolean	Validates whether the value passed is a number or not. It returns true if the parameter value is a number, and false otherwise. The method also has an optional length : Integer parameter. (To be deprecated, use Util.checkNumeric() and Util.checkNumberConstraints() instead.)
maxLength(Object value, Integer length)	Boolean	Validates whether the value passed is less than the length specified. Used to validate the length of parameters like strings and numbers. It returns true if the value passed is less than length specified, false otherwise.
numberFormat(Object value, Integer totalLength, Integer decimalLength)	Boolean	Validates whether the number represented by the value parameter is less than totalLength parameter and number of decimal digits is less than decimalLength specified. It returns true if the value passed is less than length specified in terms of both total length and length of the decimal digits, false otherwise.
setClassName(String className)	Void	Sets the custom CSS classnames on the form. The value may be null, a single CSS classname, or a space-separated list of CSS classnames that are applied to the root level of the form. The value replaces any previously set classname whether that was set in the model or by a previous call to setClassName().
setLocale(String locale)		Sets the value of locale used to render the form. It represents the locale, for example, "en" or "en_US".
setParameterValue(String paramName, Object paramValue)	Void	Sets the value of the parameter with the given name. The value should be either a Duration object, BOM JavaScript wrapper object or native JavaScript Boolean, Date, Number or String object, depending on the type of parameter.
validate(Boolean updateMessagePane)	Boolean	Forces validation to run on the form and all child panes and controls. Returns true if all validations return true. If updateMessagePane is true, then validation messages are displayed in the messages pane for any validation that failed. If updateMessagePane is not specified, it is treated as false.

Methods for Control Class

The table lists the methods for the Control class.

Control Class

Method	Return Value	Description
`getBackgroundColor()`	String	Returns the background color for an element.
		The color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at:
		http://www.w3.org/TR/CSS1/#color-units
		This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
`getClassName()`	String	Returns custom CSS classnames set on the control. The value may be `null`, a single CSS classname, or a space-separated list of CSS classnames that are applied to the root level of the form.
`getControlType()`	String	Returns the control type of this control (for example, `com.tibco.forms.controls.textbox`).
`getCustomComponentName()`	String	Returns the fully-qualified control type name as defined in the component library for the given control. If the control is not a custom control, the method returns null.
`getEnabled()`	Boolean	Retrieves the enabled flag for this control.
`getFontColor()`	String	Retrieves the font color for this control. The font color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at:
		http://www.w3.org/TR/CSS1/#color-units
		This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
`getFontName()`	String	Returns the name of the font for an element.
		This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
`getFontSize()`	Number	Returns the size of the font for the element.
		This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.

Method	Return Value	Description
getFontWeight()	String	Returns the weight of the font for an element. The return value can be either "normal", or "bold". This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getForm()	Form	Returns the form to which this control belongs.
getHint()	String	Retrieves the hint for this control.
getLabel()	String	Retrieves the label for this control.
getLink()	String	Returns the URL used by a hyperlink control.
getLinkText()	String	Returns the visible text rendered by a hyperlink control.
getOptionLabels()	String[]	Returns an array of choice labels displayed by a radiogroup or optionlist control.
getOptionValues()	String[]	Returns an array of choice values offered by a radiogroup or optionlist control.
getShortLabel()	String	Retrieves the short label associated with this control. The short label property is supported only for mobile forms.
getName()	String	Returns the name of this control.
getParent()	Pane	Returns the parent pane object to which this control belongs.
getReadOnly()	Boolean	Returns the read-only state of this control.
getRequired()	Boolean	Retrieves or sets the required flag for this control.
getTabIndex()	Integer	Returns the tab index setting configured on the control, or 0 if it is not set. This is useful for custom controls that support the setting of the tab index in their HTML markup.
getUrl()	String	Returns the URL used by an image control.

Method	Return Value	Description
getValue()	Object	Retrieves the value of this control. Equivalent to f.controlname (deprecated).
		This is either a Duration object or a native JavaScript Boolean, Date, String or Number value depending on the control type. Controls configured for list editing or multi-select drop-down lists return an array of the underlying control value type. Date, Time, and DateTime controls return a Date object. Checkbox controls return a Boolean. Duration controls return a Duration object. Numeric text input controls return a Number. All others return String.
getVisible()	Boolean	Retrieves the visible flag for this control.
getVisualProperty() (deprecated in 2.0)	String	Retrieves visual properties on the control. The only property supported in versions prior to 2.x was bgColor. The value for bgColor is hexadecimal, and is the same format as for font color.
isReallyEnabled()	Boolean	The enabled setting of a control is controlled both by its own enabled property, and the enabled properties of any of its ancestors. If the parent pane of a control is disabled, then isReallyEnabled() returns false for that control. The method returns true only if it's own enabled property is true and all of its ancestor's enabled properties are set to true.
isReallyReadOnly()	Boolean	Returns the read-only state of this control.
isReallyVisible()	Boolean	The visibility of a control is controlled both by its own visibility property, and the visibility properties of any of its ancestors. If the parent pane of a control is not visible, then isReallyVisible() returns false for that control. The method returns true only if it's own visibility property is true and all of its ancestor's visibility properties are set to true.
setBackgroundColor (String color)	Void	Sets the background color for the element. The color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at: http://www.w3.org/TR/CSS1/#color-units
setClassName(String className)	Void	Sets the custom CSS classnames on the control. The value may be null, a single CSS classname, or a space-separated list of CSS classnames that are applied to the component level of the control. The value replaces any previously set classname whether that was set in the model or by a previous call to setClassName().
setEnabled(Boolean enabledFlag)	Void	Sets the enabled flag for this control.

Method	Return Value	Description
`setFocus()`	Void	Sets focus on this control.
`setFocus(Integer)`	Void	Sets focus on this control. The API has following optional parameter: • `index`: Use this parameter for controls within a grid or record pane. Sets the focus on a control instance in the row specified by the given `index`. If the specified row is not visible, scrolls the grid control to the specified row and page. If the specified row does not exist, logs a warning message and does not shift the focus. This parameter is ignored if the control is in a singleton pane (for instance vertical pane, horizontal pane, and so on). The default value is `0`.
`setFocus(Integer index, Integer item)`	Void	Sets focus on the control. The API has following two optional parameters: • `index`: Use this parameter for controls within a grid or record pane. Sets the focus on a control instance in the row specified by the given `index`. If the specified row is not visible, scrolls the grid control to the specified row and page. If the specified row does not exist, logs a warning message and does not shift the focus. This parameter is ignored if the control is in a singleton pane (for instance vertical pane, horizontal pane, and so on). The default value is `0`. • `item`: Is used for list controls and specifies the item within the list that is to receive the focus. If the specified item does not exist, logs a warning message and does not shift the focus. This parameter is ignored if the control is not a list control. The default value is `0`. The optional parameters need not be specified for controls that are in a singleton pane (for instance vertical pane, horizontal pane, and so on). For a tabbed pane, you need to activate the particular tab (See `setActiveTab()` API on pane) before calling this API on a control within the corresponding child pane.
`setFontColor(String color)`	Void	Sets the font color for this control. The font color may be either a hexadecimal value of the form `#RRGGBB`, or one of the standard W3C colors as listed at: http://www.w3.org/TR/CSS1/#color-units
`setFontName(String fontName)`	Void	Sets the name of the font for an element. The `fontName` parameter is provided as a string to specify the name of the font.

Method	Return Value	Description
setFontSize(Integer size)	Void	Sets the size of the font for an element. The `size` parameter is provided as an integer to specify the font size in points.
setFontWeight(String weight)	Void	Sets the weight of the font for an element. The `weight` parameter is provided as a string to specify the weight of the font. It can be either `"normal"`, or `"bold"`.
setHint(String hint)	Void	Sets the hint for this control.
setLabel(String label)	Void	Sets the label for this control.
setOptions(String[] values, String[] labels)	Void	Sets the choice values and labels used by a radiogroup or optionlist control. The values and labels arrays must have the same length. The values array must not contain any null elements.
setShortLabel(String shortLabel)	Void	Sets the short label to be used for this control. The short label property is supported only for mobile forms.
setRequired (Boolean requiredFlag)	Void	Sets the required flag for this control.
setValue(Object value)	Void	Sets the value rendered by this control. Date, Time, and DateTime controls expects a Date object. Multi-select drop-down lists expect an array of Strings. Checkboxes expects a Boolean. Duration controls expects a Duration object. Numeric Text Input controls expects a Number. All other controls expect a String value. If the control is configured as a list control, then it expects an array of the underlying type.
setVisible(Boolean visibleFlag)	Void	Sets the visible flag for this control. If used from an action script for a control in a grid pane, this controls the visibility of the entire column represented by this control. If you update the visibility property of a control in a grid pane using a computation action, the setting applies to each cell in the column, but does not affect the visibility of the column itself.
setVisualProperty(String propName, String propValue) (deprecated in 2.0)	Void	Sets visual properties on the control. The only property supported in versions prior to 2.x was `bgColor`. The value for `bgColor` is hexadecimal, and is the same format as for font color.
setLink(String url)	Void	Sets the URL used by a hyperlink control.

Method	Return Value	Description
`setLinkText(String text)`	Void	Sets the visible text rendered by a hyperlink control.
`setReadOnly(Boolean readOnly)`	Void	Sets the read-only state of the control. This differs from setting the control to disabled as the user can still copy the value within the control. This is only supported for `text`, `textarea`, `date`, `datetime`, `time`, and `duration` controls.
`setUrl(String url)`	Void	Sets the URL used by an image control.
`validate(Boolean updateMessagePane)`	Boolean	Forces validation to run on the control. Returns `true` if all validations for the control return true. If `updateMessagePane` is `true`, then validation messages are displayed in the messages pane for any validation that failed. If the control is not visible, the validation runs, but `updateMessagePane` is ignored. If `updateMessagePane` is not specified, it is treated as `false`.

Methods for Pane Class

The table below lists the methods for the Pane class.

These methods cannot be used to control panes that are inside of record or grid panes. Record and grid panes contain a list of values for the panes and controls that are inside of them, but the pane methods do not contain indexes to specify which pane or control to act upon. For example, the following cannot be used to control the visibility of a tabbed pane in a record pane:

```
pane.Accounts_correspondence.setVisible(false);
```

However, a computation action can be used as a workaround. For the example shown above, create a computation action on the "Visible" property of the "correspondence" pane for the "Accounts" update event, and compute the value using the `context.record` property. For example:

```
var correspVisible = true;
if (context.record != null) {
var genInfo = context.record.getGeneralInformation();
if (genInfo != null)
{ correspVisible = "PROXYHOLDER" != genInfo.getSelectType(); }

}
correspVisible;
```

This will make the "correspondence" tab invisible when "PROXYHOLDER" is selected. The `context.record` will point to that record in the record pane relative to the instance where the value is computed.

Method	Return Value	Description
addMessage(String message, String cssClasses, Control or Pane target, Integer row)	String	Adds a message at the end of the message pane and returns a message identifier, which can be used to remove the message. The parameters are: • message: is the message string that is added at the end of the message pane • cssClasses: is the space-separated list of CSS classes to allow custom styling of the message. You need to add the cssClasses string to the element containing the message. • target: is either a control or a pane to which the message is targeted. If specified, renders a message as a link, to allow users to navigate directly to the target of the message. If null, then the message is not rendered as a link. • row: is the row of the list control or the control in a grid pane, to which the message is targeted. This is used only if a control is specified in the target parameter, and it is a list control or the control is in a grid pane. This is an optional parameter. If null, then the first element in the list control or grid pane column is targeted with a clickable message. If the target is a list control within a grid pane, then an array of length two needs to be specified. The first number in the array indicates the row of the control. The second value indicates the index of the value within the list control that receives the focus.
cancel()	Void	Cancels the modal dialog and triggers a cancel event.
clearMessages()	Void	Clears all messages added to the message pane using the addMessage() API.
close()	Void	Closes the modal dialog and triggers a close event.
getActiveTab()	Pane	Returns the active child pane for a tabbed pane.
getBackgroundColor()	String	Returns the background color for an element. The color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at: http://www.w3.org/TR/CSS1/#color-units This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getControls()	Control[]	Returns an array of controls that are direct children of this pane.
getEnabled()	Boolean	Retrieves the enabled flag for this pane.

Method	Return Value	Description
getFontColor()	String	Retrieves the font color for this pane. The font color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at: http://www.w3.org/TR/CSS1/#color-units This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getFontName()	String	Returns the name of the font for an element. This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getFontSize()	Number	Returns the size of the font for the element. This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getFontWeight()	String	Returns the weight of the font for an element. The return value can be either "normal", or "bold". This method will return the property value previously set via the corresponding set method. This will not return the original value set in the designer, which is only available in the form CSS.
getForm()	Form	Returns the form object to which this pane belongs.
getLabel()	String	Retrieves the label for this pane.
getName()	String	Returns the name of the pane.
getPanes()	Pane[]	Returns an array of panes that are direct children of this pane.
getPaneType()	String	Returns the pane type of this pane (for example, "com.tibco.forms.panes.vertical").
getParent()	Pane or Form	Returns the parent pane or form object to which this pane belongs.
getReadOnly()	Boolean	Returns the read-only state of this pane.
getSelection()	List or Object	Returns the object currently selected in the grid or record pane. If the grid supports multiple selections, then this is a list that contains the selected records.

Method	Return Value	Description
getValue()	List or Object	For grid and record panes returns a list. Returns `null` or a complex object value for other pane types.
getVisible()	Boolean	Retrieves the visible flag for this pane.
getVisualProperty() (deprecated in 2.0)	String (Hexadecimal)	Retrieves visual properties on the pane. The only property supported in versions prior to 2.x was `bgColor`. The value for `bgColor` is hexadecimal, and is the same format as for font color.
isOpen()	Boolean	Returns `True` if the modal dialog is open.
isReallyEnabled()	Boolean	The enabled setting of a pane is controlled both by its own enabled property, and the enabled properties of any of its ancestors. If the parent pane of a pane is disabled, then `isReallyEnabled()` returns `false` for that control. The method returns `true` only if its own enabled property is `true` and all of its ancestor's enabled properties are set to `true`.
isReallyReadOnly()	Boolean	Returns the read-only state of this pane.
isReallyVisible()	Boolean	The visibility of a pane is controlled both by its own visibility property, and the visibility properties of any of its ancestors. If the parent pane of a pane is not visible, then `isReallyVisible()` returns `false` for that control. The method returns `true` only if it's own visibility property is `true` and all of its ancestor's visibility properties are set to `true`.
open()	Void	Opens the modal dialog and triggers an open event. Throws an exception if the same or another modal dialog is already open on the form.
removeMessage(String messageId)	Void	Removes a message previously added to the message pane using the `addMessage()`API. You may only remove the messages added using the `addMessage()` API. The `messageId` is an identifier used to specify which message needs to be removed.
setActiveTab(Pane childPane)	Void	Sets the active child pane for a tabbed pane. The tab to be set as active tab is passed as `childPane` parameter to the method. The `childPane` parameter must be a direct child pane of the tabbed pane.
setBackgroundColor (String color)	Void	Sets the background color for the element. The `color` parameter is provided as a String in the form #RRGGBB, where RR, GG, and BB are hexadecimal numbers representing the red, blue, and green components respectively.

Method	Return Value	Description
setEnabled(Boolean enabledFlag)	Void	Sets the enabled flag for this pane.
setFontColor(String color)	Void	Sets the font color for this pane. The font color may be either a hexadecimal value of the form #RRGGBB, or one of the standard W3C colors as listed at: http://www.w3.org/TR/CSS1/#color-units
setFontName(String fontName)	Void	Sets the name of the font for an element. The fontName parameter is provided as a string to specify the name of the font.
setFontSize(Integer size)	Void	Sets the size of the font for an element. The size parameter is provided as an integer to specify the font size in points.
setFontWeight(String weight)	Void	Sets the weight of the font for an element. The weight parameter is provided as a string to specify the weight of the font. It can be either "normal", or "bold."
setLabel(String label)	Void	Sets the label for this pane.
setReadOnly()	Boolean	Sets the read-only state of this pane.
setSelection(List selection | Object selection)	Void	Valid only for grid and record panes. Sets the selected row or record of the pane to the object passed into the method. Passing null or an empty list clears the selection. If the selection is not present in the list managed by the pane, then this has no effect.
setValue(List value | Object value)	Void	Sets the value bound to the pane. For grid panes and record panes, this is a list of objects that represents either the rows or records represented by the panes. Other pane types pass a single value.
setVisible(Boolean visibleFlag)	Void	Sets the visible flag for this pane.
setVisualProperty(String propName, String propValue) (deprecated in 2.0)	Void	Sets visual properties on the pane. The only property supported in versions prior to 2.x was bgColor. The value for bgColor is hexadecimal, and is the same format as for font color.
validate(Boolean updateMessagePane)	Boolean	Forces validation to run on the pane and all child panes and controls. Returns true if all validations return true. If updateMessagePane is true, then validation messages are displayed in the messages pane for any validation that failed. If updateMessagePane is not specified, it is treated as false

Methods for List Class

The table lists the methods for the List class.

List Class

Method	Return Value	Description
`add(Object element)`	Boolean	Adds the specified element to the end of the list. If the element already exists on the list, an exception is thrown.
`add(Integer Index, Object element)`	Void	Inserts the element at the specified index. If the element already exists on the list or if the index is out of range, an exception is thrown.
`addAll(List additions)`	Void	Appends all the elements in the specified list to the end of the list.
`clear()`	Void	Removes all the elements from the list.
`contains(Object element)`	Boolean	Returns `true` if the specified element is part of the list, and `false` otherwise.
`get(Integer index)`	Object	Returns the element at the given index.
`isEmpty()`	Boolean	Returns `true` if there are no elements in the list, and `false` otherwise.
`iterator()`	Iterator	Returns an iterator over the elements in the list in proper sequence. This can be used to iterate over the items of the given list. The API methods supported by the Iterator class are listed in the table Logger Class .
`remove(Integer index)`	Boolean	Removes the element at the specified index from the list. If the index is out of range, an exception is thrown.
`remove(Object element)`	Boolean	Removes the first occurrence of the specified element from the list, if it is present.
`set(Integer index, Object element)`	Object	Replaces the element at the specified index in the list with the new specified element. If the index is out of range, an exception is thrown.
`size()`	Integer	Returns the number of elements in the list.
`subList(Integer fromIndex,Integer toIndex)`	List	Returns a list over a subset (between the specified `fromIndex`, inclusive, and `toIndex`, exclusive) of items from the original list. The sublist is backed by the original list, so changes to the sublist is reflected in the original list and vice-versa until the original list is structurally modified.

Methods for Iterator Class

The table lists the methods for the Iterator class.

Iterator Class

Method	Return Value	Description
add(Object element)	Void	Inserts the specified element into the list immediately before the element that would be returned by next(), if any, and after the element that would be returned by previous(), if any. If the element already exists on the list, an exception is thrown.
hasNext()	Boolean	Returns true if the list iterator has more elements when traversing the list in the forward direction.
hasPrevious()	Boolean	Returns true if the list iterator has more elements when traversing the list in the reverse direction.
next()	Object	Returns the next element in the list. Returns null if the iteration does not have a next element.
nextIndex()	Integer	Returns the index of the element that would be returned by a subsequent call to next(). Returns list size if the iterator is at the end of the list.
previous()	Object	Returns the previous element in the list. Returns null if the iteration does not have a previous element.
previousIndex()	Integer	Returns the index of the element that would be returned by a subsequent call to previous() or -1 if iterator is at beginning of list.
remove()	Void	Removes from the list the last element that was returned by next() or previous(). This method can be called *only* if either next() or previous() have been called and there were no calls to add() or remove() after the last call to next() or previous().

Method	Return Value	Description
`set(Object element)`	Void	Replaces the last element returned by `next()` or `previous()` with the specified element. This method can be called only if either `next()` or `previous()` have been called and there were no calls to `add()` or `remove()` after the last call to `next()` or `previous()`.

Methods for Logger Class

The table lists the methods for the Logger class.

Logger Class

Method	Return Value	Description
`fatal(String message)`	Void	Logs the given messages at the `fatal` logging level.
`error(String message)`	Void	Logs the given messages at the `error` logging level.
`warn(String message)`	Void	Logs the given messages at the `warn` logging level.
`info(String message)`	Void	Logs the given messages at the `info` logging level.
`debug(String message)`	Void	Logs the given messages at the `debug` logging level.
`trace(String message)`	Void	Logs the given messages at the `trace` logging level.
`isFatalEnabled()`	Boolean	Checks whether the `Fatal` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.
`isErrorEnabled()`	Boolean	Checks whether the `Error` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.
`isWarnEnabled()`	Boolean	Checks whether the `Warn` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.
`isInfoEnabled()`	Boolean	Checks whether the `Info` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.
`isDebugEnabled()`	Boolean	Checks whether the `Debug` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.

Method	Return Value	Description
`isTraceEnabled()`	Boolean	Checks whether the `Trace` logging level is enabled. It returns `true` if the logging level is enabled, and `false` otherwise.

Complex Data

The classes defined in the business object model (BOM) are represented in the runtime JavaScript API.

Parameters can be defined as external references to classes defined in a BOM, either in the same project as the form or in dependent projects. Each attribute defined in a class results in corresponding `get<attribute-name>()` and `set<attribute-name>(value)` methods, where `<attribute-name>` is the BOM attribute name with its first character forced to uppercase. For example, a BOM class **Customer** that had a **firstName** attribute would generate a **Customer** class with the methods:

```
String getFirstName();
void setFirstName(String firstName);
```

Factories

For each package defined in the business object model in the project, or in any projects upon which the project depends, there is an instance of a factory available in the form script editors. These are accessed via the available factory variable in each script.

New instances of complex types are created via the use of these factories. Each factory has a set of static methods that can be used to create instances of the classes defined within that package.

The factory for each package is referenced via an instance variable of the form: `factory.<business-object-model-name>`, where `<business-object-model-name>` is the fully qualified BOM package name, with "`.`" replaced by an underscore "`_`" and its first character forced to uppercase.

A `create` method for each class is provided with the signature:

```
<class-name> create<class-name>([json | object])
<class-name> listCreate<class-name>([json | object])
```

For example, suppose the package `com.example.customer` contains classes for Customer and Address. We would create instances of each of these objects using the following:

```
var address = factory.com_example_customer.createAddress();
var customer = factory.com_example_customer.createCustomer();
```

A factory is only available for packages that directly contain class definitions. For example, there would not be a factory for the `com.example` package if there were no classes defined directly in that package.

The content assist in any script editor displays the available factories after typing `factory`. Only factories for packages in the current project or referenced projects is displayed.

Method	Return Value	Description
`create<bom-class-name>([json | object])`	An instance of the given class	If no argument is provided to the method, a new object of the given type is returned. You can provide an optional argument that provides either a JSON string representation of an object of the given type, or a JavaScript object containing the same data. If you provide a wrong type of JSON or an array, the method throws a runtime exception.

Method	Return Value	Description
`listCreate<bom-class-name>([json | object])`	A List instance that contains objects of the given class	If no argument is provided to the method, an empty list is returned. If you provide a JSON specification for a single object, the method returns that object wrapped in a list. You can provide an optional argument that provides either: • a JSON String representation of an object of the given type • a JavaScript object containing the same data • a JSON String representation of an array of objects of the given type • a JavaScript array of objects representing the JSON data If you provide a wrong type of JSON, the method throws a runtime exception.

In the above methods, `<bom-class-name>` is the fully-qualified, dot-separated name of the BOM class to be instantiated. For example, to instantiate a class `MyClass` in the BOM package `com.example.test`, you need a `"$type":"com.example.test.MyClass"` attribute in the JSON, along with the values of any attributes defined for class `MyClass`.

Packages

There is also a corresponding package instance available in the `pkg` variable.

The naming of the instance of this package follows the same rule as for factories.

Enumerations defined in the package can be retrieved as read-only arrays using a method with the pattern `get[enumName]()`. So, for example, if the `com.example.customer` package contains an enumeration named **ServiceLevel** that contains the Enum Literals GOLD, SILVER, BRONZE, then you could access an array containing these three values using:

```
pkg.com_example_customer.getServiceLevel()
```

DateTimeUtil Factory

There is a built-in factory named `DateTimeUtil`. This factory provides access to methods used in creating **Duration** objects.

Method	Return Value	Description
`createDuration(Boolean isPositive, Integer years, Integer months, Integer days, Integer hours, Integer minutes, Integer seconds)`	Duration	Creates a `Duration` object that can then be used to set as a value on duration controls, the value on parameters or data fields of type Duration, or as attributes of complex objects with type Duration. Example usage: ```var duration = factory.DateTimeUtil.createDuration(true, 1,3,12,12,32,20);```

Duration Class

The value expected by the `setValue` of the duration control is an object of type Duration. The table lists the methods for the Duration class.

Method	Return Value	Description
`getYears()`	Integer	Returns the number of years set in duration.
`setYears(Integer years)`	Void	Sets number of years in duration.
`getMonths()`	Integer	Returns number of months set in duration.
`setMonths(Integer months)`	Void	Sets number of months in duration.
`getDays()`	Integer	Returns number of days set in duration.
`setDays(Integer days)`	Void	Sets number of days in duration.
`getHours()`	Integer	Returns number of hours set in duration.
`setHours(Integer hours)`	Void	Sets number of hours in duration.
`getMinutes()`	Integer	Returns number of minutes set in duration.
`setMinutes(Integer minutes)`	Void	Sets number of minutes in duration.
`getSeconds()`	Integer	Returns number of seconds set in duration.
`setSeconds(Integer seconds)`	Void	Sets number of seconds in duration.
`getMilliseconds()`	Integer	Returns number of milliseconds set in duration.
`setMilliseconds(Integer milliseconds)`	Void	Sets number of milliseconds in duration.
`isNegative()`	Boolean	Returns true if this duration is a negative duration.
`setIsNegative(Boolean isNegative)`	Void	Sets whether this duration should be treated as a negative duration.
`convert(Boolean years, Boolean months, Boolean days, Boolean hours, Boolean minutes, Boolean seconds)`	Void	Converts the internal state of the Duration object to use only the intervals specified as `true`.
`equals(Duration duration)`	Boolean	Returns `true` if the duration passed in is equal to this duration.

Method	Return Value	Description
`compareTo(Duration duration)`	Integer	Returns a negative integer, zero, or a positive integer depending on whether this object is less than, equal to, or greater than the specified duration.
`add(Date originalDate)`	Date	Returns a new date that is the result of adding the duration to `originalDate`.
`toString()`	String	Returns the duration in an ISO-8601 string.

Utility Methods

The table lists the methods for the Util class.

Util Class

Method	Return Value	Description
`tibco.forms.Util.checkDateFormat (String value)` • `value`: string containing the date value	Boolean	Checks whether the date passed as a string is in the forms date edit format (that is, ISO-8601 date format) or not. It returns `true` if the date is in the required edit format, and `false` otherwise.
`tibco.forms.Util.checkDateTimeFormat (String value)` • `value`: is a string date-time value	Boolean	Checks whether the date-time passed as a string is in the forms date-time edit format (that is, ISO-8601 date-time format) or not. Returns `true` if the date-time is in the required edit format, and `false` otherwise.
`tibco.forms.Util.checkInteger(String value)` • `value`: is a string value to be checked	Boolean	Checks whether the specified value is a valid integer or not. Returns `true` if the value is a valid integer.
`tibco.forms.Util.checkLowerLimit (String value, String lowerLimit, Boolean lowerLimitInclusive)` • `value`: is a string value to be checked • `lowerLimit`: is a string value specifying the lower limit • `lowerLimitInclusive`: is a boolean value. If `true`, the `lowerLimit` is inclusive.	Boolean	Checks whether the `value` is numerically greater than `lowerLimit`, or if `lowerLimitInclusive` is `true`, greater than or equal to `lowerLimit`. Returns `true` if the value satisfies the lower limit constraint.

Method	Return Value	Description
`tibco.forms.Util.checkMultiplicity (Object value, Integer lowerBound, Integer upperBound)` • `value`: is an object (array or list) to be checked • `lowerBound`: is an integer value specifying the lower bound • `upperBound`: is an integer value specifying the upper bound. Set `upperBound` to `-1` to signify an unbounded object.	Boolean	Checks whether a multi-valued object (array or list) has at least `lowerBound` and at most `upperBound` elements. Returns `true` if the constraints are satisfied.
`tibco.forms.Util.checkNumberConstraint (Object value, Integer totalLength, Integer decimalLength)` • `value`: is an object to be validated • `totalLength`: is an integer value specifying the maximum number of digits • `decimalLength`: is an integer value specifying the maximum number of digits following the decimal place	Boolean	Validates whether the `value` parameter has no more than `totalLength` digits and at most `decimalLength` digits following the decimal place. Returns `true` if the `value` conforms to both `totalLength` and `decimalLength` constraints.
`tibco.forms.Util.checkNumeric(String value)` • `value`: is a string value to be checked	Boolean	Checks whether the specified value is a valid number or not. Returns `true` if the value is a valid number.
`tibco.forms.Util.checkRegExp(String value, RegExp regExp)` • `value`: is a string value to be tested against `regExp` • `regExp`: JavaScript `RegExp` object with which the value is to be tested	Boolean	Validates a `value` against a regular expression. Returns `true` if the `value` matches `regExp`.
`tibco.forms.Util.checkTextLength (String value, Integer length)` • `value`: is a string value to be validated for length • `length`: is an integer value specifying the maximum length	Boolean	Checks whether the specified `value` conforms to the given `length` constraint. Returns `true` if the length of the specified `value` is less than or equal to the given `length`.

Method	Return Value	Description
`tibco.forms.Util.checkTimeFormat (String value)` • `value`: string containing the time value	Boolean	Checks whether the time passed as a string is in the forms time edit format (that is, ISO-8601 time format) or not. Returns `true` if the time is in the required edit format, and `false` otherwise.
`tibco.forms.Util.checkUpperLimit (String value, String upperLimit, Boolean upperLimitInclusive)` • `value`: is a string value to be checked • `upperLimit`: is a string specifying the upper limit • `upperLimitInclusive`: is a boolean value. If `true`, the `upperLimit` is inclusive.	Boolean	Checks whether the value is numerically less than `upperLimit`, or if `upperLimitInclusive` is `true`, less than or equal to `upperLimit`. Returns `true` if the value satisfies the upper limit constraint.
`tibco.forms.Util.compare(String value1, String value2)` • `value1`: first value to compare • `value2`: second value to compare	Integer	Compares two strings lexicographically and returns an integer that represents the comparison between the values: • returns < 0: `value1` less than `value2`. • returns 0: `value1` equal to `value2`. • returns > 0: `value1` greater than `value2`. This method is for use by the custom control wrappers only and is not supported in JavaScript Editor.
`tibco.forms.Util.copy(Object arg)` • `arg`: the BOM object you want to copy	copied object	Creates and returns a deep copy of the BOM object
`tibco.forms.Util.copyAll(List list)` • `list`: the list of BOM objects you want to copy	copied list	Creates and returns a new list containing deep copies of the BOM objects in the original list

Method	Return Value	Description
`tibco.forms.Util.escapeHtml(String text)` • `text:` text to be escaped	String	Escapes HTML markup in the given text value so it can safely be embedded in the browser without the content being interpreted as HTML. Returns the escaped text value as a string. This method is for use by the custom control wrappers only and is not supported in JavaScript Editor.
`tibco.forms.Util.formatDate (String formatString, Object|String date)` • `formatString:` a format string that conforms to the Java date format syntax, as used in the Forms framework • `date:` either a Date object or a string that conforms to the ISO-8601 date format	String	Formats a date value according to the input format. Returns a formatted date value as a string. This method is for use by the custom control wrappers only and is not supported in JavaScript Editor.
`tibco.forms.Util.formatNumber (String formatString, Number|String number)` • `formatString:` a format string that conforms to the Java number format syntax, as used in the Forms framework • `number:` a JavaScript number or a string containing a numeric value	String	Formats a number object according to the input value. Returns a formatted number as a string. This method is for use by the custom control wrappers only and is not supported in JavaScript Editor.
`tibco.forms.Util.loadCSS()` • `url:` URL to the CSS resource. Relative URLs are resolved relative to the formsclient module.	URL	Loads the Bootstrap.
`tibco.forms.Util.substitute (String template, Object args)` • `template:` containing substitution variables of the form {0}, {1}, .. {n} • `args:` string array containing values to substitute into the template. The first value in the array replaces {0} in the template, the second replaces {1}, and so on.	String	Substitutes arguments into a string template. This is useful when generating markup for controls that need an initial DOM structure before being instantiated. This is common with libraries such as jQuery or YUI. Returns a string with values substituted in the template. This method is for use by the custom control wrappers only and is not supported in JavaScript Editor.

Tips and Tricks

This chapter contains tips for working with TIBCO Forms.

Recommendations for Forms Modeling

A few good practices generally give the best results when modeling large and complex forms.

- During the early stages of form modeling, work with the labels, but don't resize panes or controls (excepting tabbed panes). Accept the default sizes until they are positioned correctly relative to each other.

  Better advice is to leave the panes to size themselves automatically. Only set an explicit size if there is a compelling reason to do so.

- Assign meaningful pane, control, parameter, action, and rule names before creating bindings and other scripts.

Grouping Related Controls Together in Vertical Panes

When a form is first generated, it contains one large pane with all the controls for the selected user task parameters.

In addition, the form contains a message pane (for error messages) and a navigation pane (for the **Cancel**, **Close**, and **Submit** buttons). These objects are created with default settings that do not normally need to be modified.

Begin by organizing large areas. Don't worry initially about configuring individual panes and controls. Concentrate on putting controls into panes with other, related controls. The positioning of the panes can best be done after this step is accomplished.

Procedure

1. Create a vertical pane for each group of related controls. Do not nest this pane inside another pane. Don't worry about multiple columns initially.

2. Give each pane a label, based on the function of the controls it will contain.

3. Drag the controls into the pane.

4. Repeat this procedure as needed for each group of related controls.

5. Modify the labels of the controls on each pane.

The Visibility Property to Simplify User Experience

If you expect to have a number of controls that are irrelevant to certain users or not applicable in certain situations, group these fields together in a vertical pane.

You can set the visibility property of this pane to false conditionally. The condition could be determined by another control. For example, a pane containing a set of controls for previous order information could be made visible or invisible depending on the runtime value of a radio control.

If "Ongoing Customer" is selected, the pane labeled Previous Orders, and all of its controls, would be visible, but that pane would be invisible if "New Customer" were selected.

Configuration of the Pane Type Property (optional)

If desired, change vertical panes to horizontal or tabbed panes by configuring the **Pane Type** property on the **General** tab of the pane's Properties View.

Modifying Excessively Long Forms

An extremely long form requires unwanted scrolling to be viewed or filled in and adversely affects the user's experience.

Procedure

1. Create multiple columns.
   a) Place groups of controls into two or more separate vertical panes, each representing a separate column. Drag the second pane to a position next to the first pane, so that you see a dotted line appear. The dotted line means that a horizontal pane will be created to contain the two vertical panes.
   b) If you want more than two vertical columns, drag additional panes, one at a time, next to the right-most vertical pane within the new horizontal parent pane.
   c) If you want to have other groups of fields in the second column, rather than adding another column next to the existing columns, you will probably want to first create an additional horizontal pane to hold them. Then place new vertical panes within this horizontal parent pane, and add the additional controls to the vertical panes. This ensures that the added controls are aligned vertically and horizontally.

      You can create, for instance, two columns out of four modules by first creating two horizontal panes, one above the other, and placing two vertical panes inside each of them. Your controls would be place within the four vertical panes.

      In this way, the default tab sequence will be left-to-right, and then top-to-bottom, the way you read a book, which is the tabbing behavior expected by most users.

2. Use tabbed panes in place of vertical panes. See the section, Creation of Tabbed Panes.

## Expansion of Narrow Panes to Avoid Wrong Placement at Run Time

When there are two or more narrow root-level panes with a combined fixed width that is less than the available space into they are rendered, at runtime such panes may be rendered next to each other instead of on top of each other.

To avoid improper pane placement in this case, you can do one of the following:

- Increase the width of these panes so that there is not enough room left for them to be rendered side by side

- Set the overflow attribute for the panes to expand, so that the panes expand and fill the available space.

Creation of Tabbed Panes

Use tabbed panes only when there is information that is seldom used. Because they are partially hidden and can be hard to find, user interface specialists often recommend that they be avoided or used cautiously.
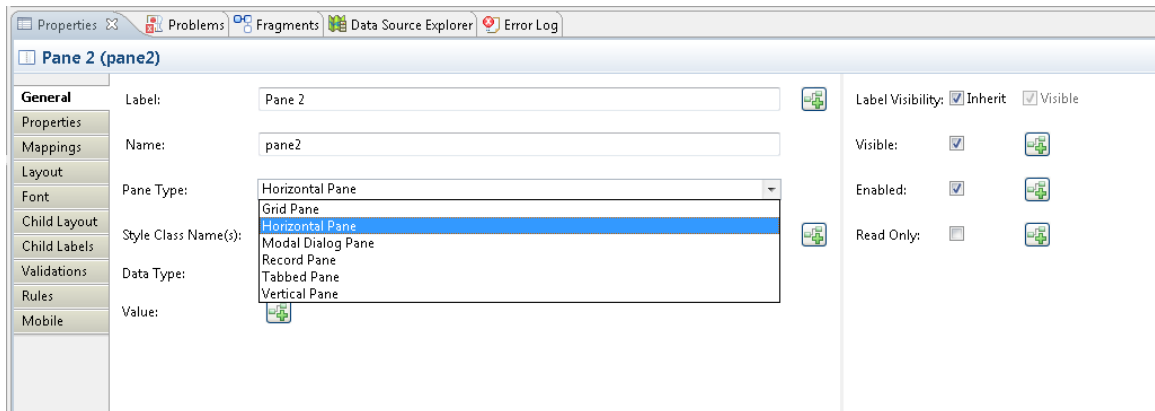
Addition of a Tab to an Existing Tabbed Pane

If you want to add a tab to an existing tabbed pane, click the button for adding a new child pane. This button is circled in the figure New Child Pane Button:

New Child Pane Button
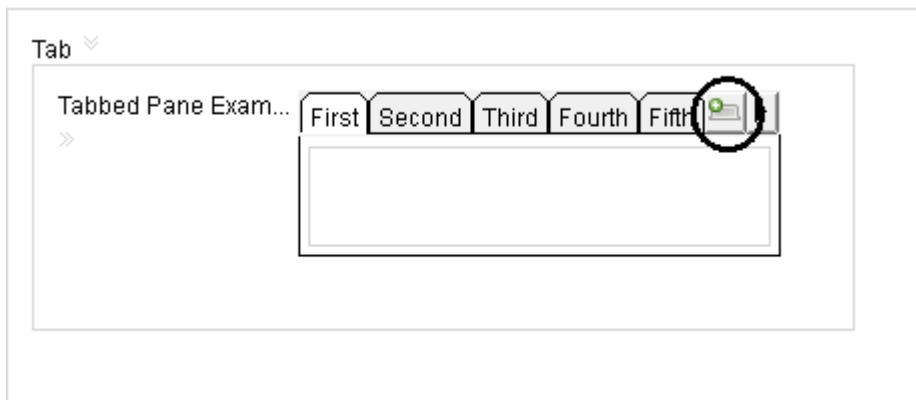
Vertical and horizontal panes are automatically resized so that the panes or controls they contain fit properly.

FAQs on TIBCO Business Studio Forms

You may benefit from the answers to a few frequently asked questions, which can go a long way in enhancing your forms.

How can I create a Form that is not associated with any business process?

Right-click **Forms** in the Project Explorer, click **New**, and click **Form**. Alternatively, on the **File** menu, click **New** > **Other** to open the New dialog box. Expand **TIBCO Forms** by clicking the plus icon, and click **Form**.

How can I create multiple columns of controls on a Form?

Create a horizontal pane, and place two or more vertical panes inside of it. Then add controls to the vertical (child) panes. Each vertical pane will contain a column of controls.

How can I align the labels between different panes, for example, when there are several panes that are direct children of a root pane?

To align the label width for all panes to achieve a uniform and consistent appearance, set an explicit child label width for all panes whose labels should be aligned.

How can I create option lists or radio buttons using array type parameters?

In the properties tab of the option list or radio button Properties View, create **Label Array** and **Value Array** Choice Bindings. If different parameters are selected for **Values** and **Labels,** you must ensure that the number of items in both the arrays are equal.

How can I reuse similar behavior between different controls?

Do not reference the control by name in your shared action scripts, but rather use the *context.control* object that represents the source control of the fired event.

How can I use part of the parameter value as a value for a control?

While capturing input values for items like social security numbers, different controls can be used to capture different parts of the same value.

For example a value of parameter ssn, say, 888-78-9898, can be captured in three text fields. First, a text control named ssn_part1 takes input for the first part, 888.

Second, a text control called ssn_part2 takes input for the second part, 7. Finally, a text control called ssn_part3 takes input for the last part 9898.

This can be achieved by providing a scriptlet that returns different parts of the parameter value. In this example the three expressions would be p.ssn.substring(0,3), p.ssn.substring(5,7), p.ssn.substring(8,12). Each of these scripts would be provided in their own computation actions within a rule that fires when the underlying parameter is updated.

Similarly, a scriptlet that assembles the values of 3 controls can be used as an expression for a parameter.

In this example this expression would be:

```
f.ssn_part1 + "-" + f.ssn_part2 + "-" + f.ssn_part3;
```

Tips for Using TIBCO Business Studio Forms

You may benefit from a few tips.

If possible, reproduce problems in Forms Preview

It is much quicker to track down problems if an issue can be reproduced within the Forms preview. Alter the logging level in the preview from **Window** > **Preferences** > **Form Designer** > **Preview**.

Determine source of data integrity issues

For any issues with the initial data being displayed in a form, you can use the runtime logging to see the data being received by the form and the data being submitted. To see this, enable INFO level logging by adding the query parameter log_level=INFO to the Openspace or Workspace URL. For example, http://<host>:<port>/openspace/openspace.html?log_level=INFO.

You can see the logging messages that show the data provided to the form on Openspace, and the data submitted back to the server side. This helps in tracking down issues where you see unexpected results in subsequent steps. If the data coming into and out of a form is correct, then there may be some issues in other tasks in the process or in the page-flow.

Make use of sample data in TIBCO Business Studio to reproduce problems at runtime

You can use the INFO logging to capture actual runtime input data in order to reproduce problems within Forms preview.

You can follow these steps:

1. From the logging window, copy the incoming data from the log message. For example, `{items: [ . . . . . . . . ]}`

2. In TIBCO Business Studio, create a file in the same folder as the form, giving it a `.data.json` extension. For example, `myCustomData.data.json`

3. On the Preview Data property sheet of the form, select **Custom**, and select the newly created file from the drop-down list.

Now when you preview the form, this data is used as the initial data for the form. With this setting, you can reproduce and troubleshoot data-specific issues within TIBCO Business Studio without having to re-deploy the projects.

Use logging within form actions

You can access an instance of a logger class using `context.form.logger`. There are methods for logging at TRACE, DEBUG, INFO, WARN, and ERROR levels. For example, `logger.debug("My debug message")`.

Temporarily disable actions and rules

If you suspect a problem in your action script, you can disable the rules and actions using the TIBCO Business Studio UI. It can help in tracking down the problem action.

Check for null objects and values

Problems often arise in scripts that are accessing values before they have been initialized.

Make sure the Form parameter interface is in sync with the User Task

Sometimes the reference from the form to the user task is lost, in which case the validation framework may not report a synchronization issue. This can happen if you are using the same form for multiple user tasks (not generally recommended), or have refactored or moved resources within or between projects.