Widgets Reference

Customizable SendMessage Registration Form

4/2/2025

Contents

Customizable SendMessage Registration Form

Introduced: 9.0.014.03

SendMessage allows you to customize the registration form shown to users prior to starting a session. The following form inputs are currently supported:

- Text
- Select
- Hidden
- Checkbox
- Textarea

Customization is done through an object definition that defines the layout, input type, label, and attributes for each input. You can set the default registration form definition in the `_genesys.widgets.sendmessage.form` configuration option. Alternately, you can pass a new registration form definition through the `SendMessage.open` command:

```
_genesys.widgets.bus.command("SendMessage.open", {formJSON: oRegFormDef});
```

Inputs are rendered as stacked rows with one input and one optional label per row.

Default Example

The following example is the default object used to render SendMessage's registration form. This is a very simple definition that does not use many properties.

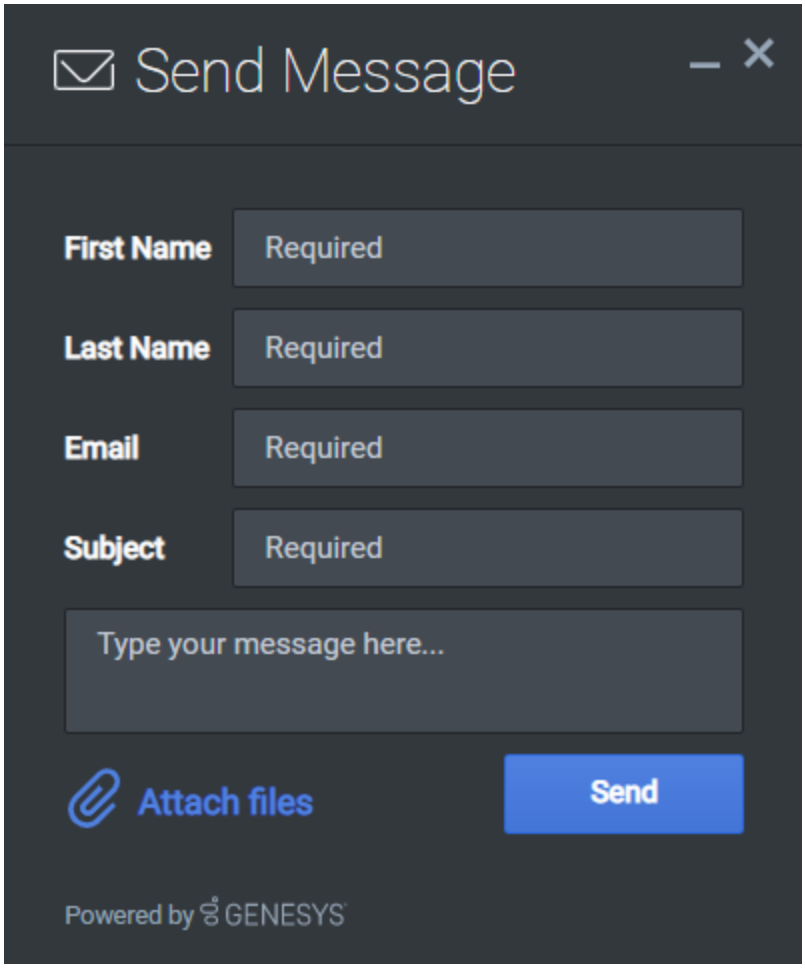
> ### Important
> You can define *any* number of inputs here, of *any* supported type, in *any* combination. Our example below simply demonstrates how SendMessage defines its default form internally.

> ### Important
> The fields with the names (firstname, lastname, email, subject, messagebody) are required for all Sendmessage custom forms. These field values are required by

> Genesys Sendmessage API to send messages.

```
{
        wrapper: "<table></table>",
        inputs: [

                {
                        id: "cx_sendmessage_form_firstname",
                        name: "firstname",
                        maxlength: "100",
                        placeholder: "@i18n:sendmessage.PlaceholderFirstName",
                        label: "@i18n:sendmessage.FirstName"
                },

                {
                        id: "cx_sendmessage_form_lastname",
                        name: "lastname",
                        maxlength: "100",
                        placeholder: "@i18n:sendmessage.PlaceholderLastName",
                        label: "@i18n:sendmessage.LastName"
                },

                {
                        id: "cx_sendmessage_form_email",
                        type: "email",
                        name: "email",
                        maxlength: "100",
                        placeholder: "@i18n:sendmessage.PlaceholderEmail",
                        label: "@i18n:sendmessage.Email"
                },

                {
                        id: "cx_sendmessage_form_subject",
                        name: "subject",
                        maxlength: "100",
                        placeholder: "@i18n:sendmessage.PlaceholderSubject",
                        label: "@i18n:sendmessage.Subject"
                },

                {
                        id: "cx_sendmessage_form_messagebody",
                        type: "textarea",
                        name: "messagebody",
                        rows: "2",
                        placeholder: "@i18n:sendmessage.PlaceholderTypetexthere",
                        label: false
                }
        ]
}
```

Using this definition will result in this output:



Properties

Each input definition can contain any number of properties. These are categorized in two groups: "Special Properties", which are custom properties used internally to handle rendering logic, and "HTML Attributes" which are properties that are applied directly as HTML attributes on the input element.

Special Properties

" "

Property	Type	Default	Description
type	string	"text"	Sets the type of input to render. Possible values are currently "text", "hidden", "select", "checkbox", and

Property	Type	Default	Description
			"textarea".
label	string		Sets the text for the label. If no value is provided, no label will be shown. You may use localization query strings to enable custom localization (for example, label: "@i18n:namespace.StringName"). Localization query strings allow you to use strings from any widget namespace or to create your own namespace in the localization file (i18n.json) and use strings from there (for example, label: "@i18n:myCustomNamespace.myCusto For more information, see the Labels section.
wrapper	HTML string	"	
{label}	{input}		
Each input exists in its own row in the form. By default this is a table row with the label in the left cell and the input in the right cell. You can redefine this wrapper and layout by specifying a new HTML row structure. See the Wrappers section for more info. The default wrapper for an input is "			
{label}	{input}		
validate	function		Defines a validation function for the input that executes when the input loses focus (blur) or changes value. Your function must return `true` or `false`. True to indicate it passed, false to indicate it failed. If your validation fails, the form does not submit and the invalid input is highlighted in red. See

Property	Type	Default	Description
			the Validation section for more details and examples.
validateWhileTyping	boolean	false	Executes validation on keypress in addition to blur and change. This ignores non-character keys like shift, ctrl, and alt.
options	array	[]	When ‘type’ is set to ‘select’, you can populate the select by adding options to this array. Each option is an object (for example, {text: ‘Option 1’, value: ‘1’} for a selectable option, and {text: "Group 1", group: true} for an option group).

HTML Attributes

With the exception of special properties, all properties are added as HTML attributes on the input element. You can use standard HTML attributes or make your own.

Example

```
{
        id: "cx_sendmessage_form_firstname",
        name: "firstname",
        maxlength: "100",
        placeholder: "@i18n:sendmessage.PlaceholderFirstName",
        label: "@i18n:sendmessage.FirstName"
}
```

In this example, `id`, `name`, `maxlength`, and `placeholder` are all standard HTML attributes for the text input element. Whatever values are set here are applied to the input as HTML attributes.

Note: the default input type is "text", so type does not need to be defined if you intend to make a text input.

HTML Output

```
<input type="text" id="cx_sendmessage_form_firstname"
 name="firstname" maxlength="100" placeholder="Required"></input>
```

Labels

A label tag is generated for your input if you specify label text and if your custom input wrapper includes a ‘{label}’ designation. If you have added an ID attribute for your input, the label automatically links to your input so that clicking on the label selects the input or, for checkboxes, toggles it.

Labels can be defined as static strings or localization queries.

Wrappers

Wrappers are HTML string templates that define a layout. There are two kinds of wrappers, **Form Wrappers** and **Input Wrappers**:

Form Wrapper

You can specify the parent wrapper for the overall form in the top-level "wrapper" property. In the example below, we specify this value as "

". This is the default wrapper for the SendMessage form.

```
{
        wrapper: "<table></table>", /* form wrapper */
        inputs: []
}
```

Input Wrapper

Each input is rendered as a table row inside the Form Wrapper. You can change this by defining a new wrapper template for your input row. Inside your template you can specify where you want the input and label to be by adding the identifiers "{label}" and "{input}" to your wrapper value. See the example below:

```
{
        id: "cx_sendmessage_form_firstname",
        name: "firstname",
        maxlength: "100",
        placeholder: "@i18n:sendmessage.PlaceholderFirstName",
        label: "@i18n:sendmessage.FirstName",
        wrapper: "<tr><th>{label}</th><td>{input}</td></tr>" /* input row wrapper */
}
```

The {label} identifier is optional. Omitting it allows the input to fill the row. If you decide to keep the label, you can move it to any location within the wrapper, such as putting the label on the right, or stacking the label on top of the input. You can control the layout of each row independently, depending on your needs.

You are not restricted to using a table for your form.You can change the form

wrapper to "
" and then change the individual input wrappers from a table row to your own specification. Be aware though that when you move away from the default table wrappers, you are responsible for styling and aligning your layout. Only the default table row wrapper is supported by default Themes and CSS.

Validation

You can apply a validation function to each input that lets you check the value after a change has been made and/or the user has moved to a different input (on change and on blur). You can enable validation on key press by setting `validateWhileTyping` to `true` in your input definition.

Here is how a validation function is defined:

```
{
        id: "cx_sendmessage_form_firstname",
        name: "firstname",
        maxlength: "100",
        placeholder: "@i18n:sendmessage.PlaceholderFirstName",
        label: "@i18n:sendmessage.FirstName",

        validateWhileTyping: true, // default is false

        validate: function(event, form, input, label, $, CXBus, Common){

                return true; // or false
        }
}
```

You must return `true` or `false` to indicate that validation has passed or failed, respectively. If you return false, the SendMessage form will not submit, and the input will be highlighted in red. This is achieved by adding the CSS class "cx-error" to the input.

Validation Function Arguments

Argument	Type	Description
event	JavaScript event object	
form	HTML reference	A jquery reference to the form wrapper element.
input	HTML reference	A jquery reference to the input element being validated.
label	HTML reference	A jquery reference to the label for the input being validated.
$	jquery instance	Widget’s internal jquery instance. Use this to help you write your validation logic, if needed.
CXBus	CXBus instance	Widget’s internal CXBus reference. Use this to call commands on the bus, if needed.

Argument	Type	Description
Common	Function Library	Widget’s internal Common library of functions and utilities. Use if needed.

Form Submit

Custom Input field form values are submitted to the server as key value pairs under the userData section of the form submit request, where input field names will be the property keys. During the submit, this data is merged along with the userData defined in the SendMessage open command.