Integrate the GFWizardBase

URL: https://docs.grassfish.com/docs/integrate-the-gfwizardbase

Archiviert am: 2025-07-17 18:36:08

The GFWizardBase is the interface between the spot and all areas in which the spot is played and displayed. This is a JavaScript library that you must integrate into HTML Wizard spots to ensure that the spot functions properly in the Grassfish system.

You can nest the file GFWizardBase in subdirectories but don't edit it. Use a **<script>** tag to insert the file into the website.

> *Note*
>
> *You must not access the methods of the GFWizardBase until the JavaScript file has been initialized.*

To access the methods, use the **body onload** method as follows:

```
 <body onload="init()">

//gfWizardBase must be accessed globally!
var gfWizardBase = new GFWizardBase();

function init()
{
//register all events
gfWizardBase.registerDataChangedHandler(onDataChanged);
//notify the container that you are ready
gfWizardBase.sendReady();

alert(gfWizardBase.getVersion());
}

function onDataChanged(jsonData)
{

}
```

Initialize the spot

Initially the spot must send a **ready** event so that the containers know that they can now communicate with the spot. This invocation looks as follows:

gfWizardBase.sendReady();

Once the spot has reported that it's ready, it receives its JSON data from the container. Once the data has been set, the spot automatically sends an **InitComplete** event to the container which is forwarded to the player. As a result, the player knows that it can play the spot.

You can delay the **InitComplete** if needed by adding a true parameter to the **sendReady**. As a result, the **InitComplete** event is not automatically triggered upon setting the data but must be triggered manually.

```
 gfWizardBase.sendReady(true);

//do your asynchronous stuff, e.g. call a webservice

gfWizardBase.sendInitComplete();
```

Note

- If you decide to manually trigger **InitComplete**, you must do so or errors will occur during playback.
- If you also use the GFSpotBase in an HTML Wizard spot, trigger only the **sendInitComplete()** of the GFWizardBase and not of the GFSpotBase.

Receive data

For the spot to receive data filled in by a user, it must register for the **dataChanged** event. This is carried out as follows:

gfWizardBase.registerDataChangedHandler(onDataChanged);

Function	Registers a call-back function that is initially invoked and if any data has changed in the HTML Wizard.
Parameters	function(jsonData) *jsonData* the JSON data that come from the HTML Wizard (the modified content from the ascData.json file) [string]

Examples	
	```
	 gfWizardBase.registerDataChangedHandler(onDataChanged);

	function onDataChanged(jsonData)
	{
	var elementData = gfWizardBase.getElementData();
	document.getElementById("myElement").innerHTML = elementData["myJsonElement"].Value;
	}
	```

Players	Windows/Linux, Android

Start and stop animations

The spot can start and stop multiple times in the HTML Wizard. To start the animations, the container sends a play command via call-back, a stop call-back is executed to stop them.

gfWizardBase.registerPlayHandler(onPlay);

Function	Starts the animations in the spot. This is invoked by the player, the preview, or the HTML Wizard when the animations in the spot should start. Prior to this, the spot should not play any animations.
Parameters	None
Examples	gfWizardBase.registerPlayHandler(startMyAnimations);
Players	Windows/Linux, Android

gfWizardBase.registerStopHandler(onStop);

Function	Stops the animations in the spot. It is invoked by the player, the preview, or the HTML Wizard when the animations in the spot should stop. The spot should reset everything to its original status because play can be executed again in the HTML Wizard.
Parameters	None
Examples	gfWizardBase.registerStopHandler(resetMyAnimations);
Players	Windows/Linux from global version 10.0 onwards Android from player version 8.1.0.2 onwards

Log messages

It’s possible to have on-screen logging output. Messages to display here must be sent to the container via **sendLog**. You can connect to these log messages to continue to react to them in the spot via call-back **registerLogHandler**.

gfWizardBase.sendLog("message");

Function	Logs the message in the on-screen log of the asc interface. The log must be activated in the asc interface by means of the URL parameter &debug=true.
Parameters	*message* the message that is to be logged [string]
Examples	gfWizardBase.sendLog("message");

Players	Windows/Linux

gfWizardBase.registerLogHandler(onLog);

Function	Call-back function for sendLog to further process it in the spot.
Parameters	*message* the message that to log [string]

Examples	`gfWizardBase.registerLogHandler(onLog);`
	
	`function onLog(message)`
	`{`
	`}`

Players	Windows/Linux

Select an element

In the wizard, you can select an element that triggers an event in the spot. In addition, you can select elements in the spot and send an event to the wizard.

You can register for the **registerElementSelectedHandler** function to react to events through selections made in the wizard. Use the **sendSelectElement** function to trigger a selection event yourself.

gfWizardBase.registerElementSelectedHandler(onElementSelected);

Function	This call-back is invoked if an element has been selected in the wizard tree.
Parameters	function(id) *id* the ElementId of the selected element from the JSON data [string]

Examples	`gfWizardBase.registerElementSelectedHandler(onElementSelected);`
	
	`function onElementSelected(id)`
	`{`
	
	`}`

Players	Not relevant for players

gfWizardBase.sendSelectElement("id");

Function	This function tells the wizard that an element should be selected in the tree.
Parameters	*id* the ElementId of an element to be selected from the JSON data [string]
Examples	gfWizardBase.sendSelectElement("id");
Players	Not relevant for players

Set tasks

You can set tasks for HTML Wizard spots via the JSON configuration that are displayed to the user in the editor.

Note

Don’t save spots with incomplete tasks or add them to playlists. They can’t be played or transferred to the player.

With tasks, you can create dependencies among individual configurations. It’s possible to link a mandatory entry to a condition; the entry of a **SimpleText** element is only mandatory if a linked list element has a specific value.

gfWizardBase.addTodo("id", "message");

Function	With this function, it is possible to register a task. This task is assigned with a specific element in which the task is shown and supplied with text in order to describe the required action. Optionally, translations can be set to show the text of the task in different languages. Invoking the function registers a corresponding task text for a specified element. The display is not carried out yet which provides the possibility to collect multiple tasks. The display is triggered by the invocation of the sendTodos function.
Parameters	*elementId* Unique ID of the element (ascData.json) [string] *message* message which is displayed [string] *translation* translation in the format {"en": "translation"} [object]
Examples	gfWizardBase.addTodo("id", "message", {"en": "translation"}); s

Players	Not relevant for players

gfWizardBase.sendTodos();

Function	This function transfers all previously collected tasks via the addTodo function to the editor and displays them. By invoking the sendTodo function, the editor is notified that all tasks have been collected, validated, and are ready to be displayed.
Parameters	None

Examples	`gfWizardBase.addTodo("textElement1", "do something");`
	
	`gfWizardBase.addTodo("textElement2", "do something else");`
	
	`gfWizardBase.sendTodos();`

Players	Not relevant for players

Exmaple: definition of tasks

In this example, the editor should have a **Boolean** and a **SimpleText** element.

- If the **Boolean** element is inactive, no validation is required and any **SimpleText** element can be entered.
- If the **Boolean** element is activated, exactly two **SimpleText** values must be specified.

As usual, you must set both element entries via the ascData.json file and, therefore, they have no logical connection to each other at first:

```
 ...
"Elements": [
   {
      "Id": "checkbox",
      "DataType": "boolean",
      "DisplayName": "Checkbox",
      "Value": true
   },
   {
      "Id": "text",
      "DataType": "simpleText",
      "DisplayName": "Text",
      "Value": [
         {
            "Text": ""
         }
      ],
      "Options": {
         "NumberOfInstances": 5
```

```
        }
    }
]
...
```

By defining a task, you can display the desired dependency, whereas the required functions are supplied by the GFWizardBase library. Implement it as follows:

gfWizardBase.registerDataChangedHandler(function(jsonData)

```
 gfWizardBase.registerDataChangedHandler(function(jsonData)
{
   var spot = gfWizardBase.getChainedElementData();

   // validate if checked
   if (spot.checkbox.getValue())
   {

      // check length and both text fields valid
      if (spot.text.getValues().length !== 2 ||
         !spot.text.getValue(0) || !spot.text.getValue(1))
      {
         gfWizardBase.addTodo("text", "2 text values are mandatory");
      }
   }

   // propagate todos
   gfWizardBase.sendTodos();

});
```

The validation is carried out via the registered call-back function of the **dataChangedHandler** and invoked via an entry in the editor which implies a data change.

If the checkbox is active, the validation is carried out and checks if two **SimpleText** values are available. Additionally to the validation of the length of the array, both values are checked.

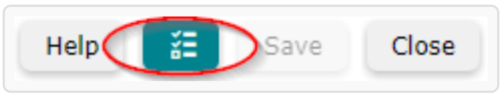
In case of an error, the task is registered via the **addTodo** function and transferred to the editor via the **sendTodo** function to be displayed.

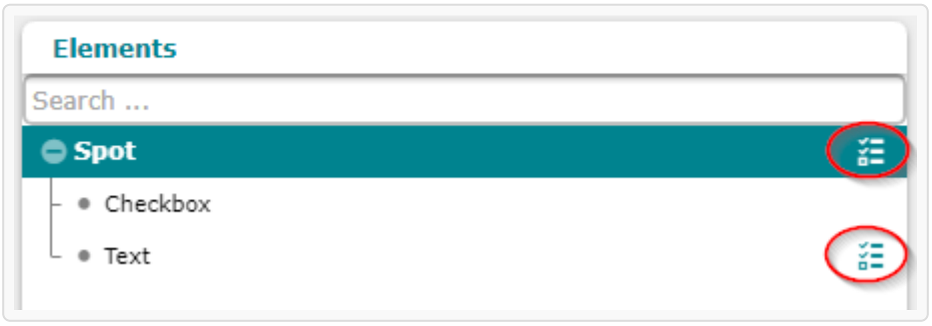
Note

It’s possible to set an unlimited number of tasks with corresponding validations. These must be registered or carried out prior to the actual transfer to the editor. That means prior to the sendTodo function.

Tasks in the editor

When you open the editor in IXM One and the spot has pending tasks, you can see the following icons:





Click on the respective icon to learn more about the tasks and to complete them.

Specify errors

In addition to the configuration via the JSON structure, you can also specify errors in HTML Wizard spots. For example, dependencies between elements that are not valid should not be allowed as entries.

Note

You can’t save spots with existing errors. The errors must be resolved first.

The operation of errors and tasks is identical, errors are however set via different functions and further communicated to the editor.

gfWizardBase.addError("id", "message");

Function	This function allows to register an error. The error is assigned to a specific element where the error is shown and provided with text to describe the required correction. Optionally, you can specify translations for error text in different languages. By invoking the function, a corresponding error text is registered for a specific element. The text is not displayed yet and therefore it is possible to collect multiple errors. The display of these errors is invoked directly via the sendErrors function.
Parameters	*elementId* Unique ID of the element (ascData.json) [string] *message* message shown [string] *translation* translation in the format {"en": "translation"} [object]
Examples	gfWizardBase.addError("id", "message", {"en": "translation"}); s
Players	Not relevant for players

gfWizardBase.sendErrors();

Function	With this function, all the previously collected tasks via the addTodo function are transferred and displayed in the editor. By invoking the sendErrors function, the editor is informed that all errors have been collected, validated, and are ready to be displayed.
Parameters	None

Examples	`gfWizardBase.addError ("textElement1", "something happened");`
	`gfWizardBase.addError ("textElement2", "another thing happened");`
	
	`gfWizardBase.sendErrors();`

Players	Not relevant for players

Use templates

Use templates to specify repeating structures within the JSON configuration that are included dynamically via the editor.

This can reduce the complexity of the structure. You can use an available template as many times as you need which increases flexibility.

Example: using templates

You can maintain a list of team information via the editor. Individual users receive an entry for the name, the function, and a photo.

Without templates, you must create a group for each employee with two SimpleText elements for the name and function as well as a media element for the photo. You must copy this group for every employee and edit the identification (Id) of the individual elements. This presents two disadvantages:

- The number of employees is fixed and can only be changed by editing the JSON files.
- If the employee structure changes, for example, with an additional attribute for emails, you must edit all incidents. Depending on the number of entries, this can be a very tedious and error-prone task.

With a template, you only have to specify the employee structure once as a template in the JSON file. Via a placeholder, it can be included as often as needed in the editor. With additional options, it’s possible to specify further properties for the placeholder:

- Selection of desired template (for example, distinction between operating and executive positions)
- Number of employees can be limited

Note

You can find examples for the use of templates and nested templates on the training website.