Widget BUS Guide

Genesys Widgets Current

2/5/2023

Table of Contents

Genesys Widgets Bus Guide

Tip

The latest version of our documentation (titled “**Current**”) relates to release **9.0.x**.

The Widget Bus API Guide provides reference information that you can use to control the Genesys WebChat Widget, including:

- Overview of the Widget Bus API
- Extensions

Widget Bus API Overview

The Widget Bus (CXBus) is a publish/subscribe/command bus designed for User Interfaces. It allows different UI components and controllers to communicate with each other and bind business logic together into a larger, cohesive product.

CXBus supports publishing and subscribing arbitrary events with data over the bus and any other plugin on the bus can subscribe the that event. Publications and subscriptions are loosely bound so that you can publish and subscribe to any event without that event explicitly being available. This allows for plugins to lazy load into the bus or provide conditional logic in your plugins that wait for other plugins to be available.

The full CXBus API reference for each Widget/Plugin is available here: Widgets Reference

CXBus events and commands are executed asynchronously using deferred methods and promises. This allows for better performance and standardized Pass/Fail handling for all commands. Command promises are not resolved until the command is finished, including any nested asynchronous commands that command may invoke. This gives you assurance that the command completed successfully and the timing of your follow-up action will occur at the right time. As for permissions, CXBus provides metadata in every command call including which plugin called the command and at what time. This allows for plugins to selectively allow/deny invocation of commands.

You can use three methods to access the Bus:

- Global access
- Genesys Widgets onReady callback
- Extensions

Global Access

QuickBus

`window._genesys.widgets.bus`
For quick access to call commands on the bus, you can access the **QuickBus** instance after Genesys Widgets loads. QuickBus is a CXBus plugin that is exposed globally for your convenience. Typical use cases for using QuickBus are for debugging or calling a command when a link or button is clicked. Instead of creating your own plugin, you can use QuickBus to add the click handler inline in your HTML.
Example:

```
<a href="#" onclick="_genesys.widgets.bus.command('WebChat.open');">Open WebChat</a>
```

Global CXBus

Starting in release 9.0.002.06, CXBus is available as a global instance named "CXBus" (or window.CXBus). Unlike QuickBus, this is not a plugin but CXBus itself.
In version 9.0.003.xx, CXBus has been updated to include a "command" method that allows you to execute a command directly from the CXBus instance.

Example:

```
CXBus.command("WebChat.open");
```

You can use this, like QuickBus, for debugging or setting up click events.

Genesys Widgets onReady callback

Genesys Widgets provides an "onReady" callback function that you can define in your configuration. This will be triggered after Genesys Widgets initializes. QuickBus is provided as an argument in this function, but you may also access CXBus globally in your function in version 9.0.002.06 or later.

```
window._genesys.widgets.onReady = function(QuickBus){

    // Use the QuickBus plugin provided here to interface with the bus
    // QuickBus is analogous to window._genesys.widgets.bus
};
```

Extensions

You can define your own plugins/widgets that interface with Genesys Widgets. For more information, please see Extensions.

CXBus Reference

Starting in release 9.0.002.06, the CXBus instance is exposed globally (window.CXBus). The CXBus instance has several methods available.

CXBus.command

[Available in 9.0.003.xx] Calls a command on the bus under the namespace "CXBus". Use this to quickly and easily call commands without needing to generate a unique plugin interface object first.

Example

```
CXBus.command("WebChat.open", {});
```

Arguments

Name	Type	Description
Command Name	string	The name of the command you wish to execute.
Command Options	object	Optional: You may pass an object containing properties that the command will accept. Refer to the documentation on each command to see what options are available.

Returns

Always returns a promise. You can define done(), fail(), or always() callbacks for every command.

CXBus.configure

Allows you to change configuration options for CXBus.

Example

```
CXBus.configure({debug: true, pluginsPath: "/js/widgets/plugins/"});
```

Arguments

Name	Type	Description
Configuration Options	object	An object containing properties, similar to command options. In this object you can change configuration options for CXBus.

Configuration Options

Name	Type	Description
debug	boolean	Enable or disable CXBus logging in the javascript console. Set to **true** to enable, set to **false** to disable. Default value is **false**.
pluginsPath	string	The location of the Genesys Widgets "plugins" folder. **Example:** `"/js/widgets/plugins/"` The default value here is "". This configuration option is used for lazy-loading plugin files. Be sure to configure this option when using Genesys Widgets in Lazy-Loading mode.
pluginMap	object	Used to change the target JS file for each plugin or to add a new plugin. **Example:** `{sendmessage: "https://www.yoursite.com/ plugins/custom-`

Name	Type	Description

		`sendmessage.js"}`
		
		CXBus will automatically lazy-load plugins defined in this object when something tries to call a command on that plugin.
		
		For instance, if `SendMessage.open` is called and **SendMessage** isn't loaded, CXBus will fetch it from the default "plugins/" folder. If you want to load a different **SendMessage** widget, you can override the default URL of the JS file associated with "sendmessage".
		
		You can also prevent a plugin from loading by mapping it to **false**.
		
		Example:
		
		`{sendmessage: false}`
		
		Important
		• Any number of plugins can be included in this object.
		• Only works when using the lazy-loading method of initializing Widgets.
		• Not intended to be used to load different versions of Genesys Widgets plugins.
		• Intended to be used along with the proper pluginsPath configuration. Do not use `pluginMap` method separately.

Returns

This method returns nothing.

CXBus.loadFile

Loads any javascript file.

Example

```
CXBus.loadFile("/js/widgets/plugins/webchat.min.js");
```

Arguments

Name	Type	Description
File Path	string	Loads a javascript file based on the file path specified.

Returns

Always returns a promise. You can define done(), fail(), or always() callbacks. When the file loads successfully, done() will be triggered. When the file fails to load, fail() will be triggered.

CXBus.loadPlugin

Loads a plugin file from the configured "plugins" folder.

Example

```
CXBus.loadPlugin("webchat");
```

Arguments

Name	Type	Description
Plugin Name	string	Loads a plugin from the "plugins" folder by name (configured by the "pluginsPath" option). Plugin names match their CXBus namespaces but are lowercase. Example: To load WebChat, use "webchat". You can refer to the files inside the "plugins" folder as well. The first part of the file name will be the name you use with this function. Example: Use "webchat" to load "webchat.min.js".

Returns

Always returns a promise. You can define done(), fail(), or always() callbacks. When the plugin loads successfully, done() will be triggered. When the plugin fails to load, fail() will be triggered.

CXBus.registerPlugin

Registers a new plugin namespace on the bus and returns a plugin interface object. You will use the plugin interface object to publish, subscribe, call commands, and perform other CXBus functions.

Example

```
var oMyPlugin = CXBus.registerPlugin("MyPlugin");
```

Arguments

Name	Type	Description
CXBus Plugin Namespace	string	The namespace you want to reserve for your plugin.

Returns

If the namespace is not already taken, it will return a CXBus plugin interface object configured with the selected namespace. If the namespace is already taken, it will return false.

CXBus Plugin Interface Reference

When you register a plugin using CXBus.registerPlugin(), it returns a CXBus Plugin Interface Object. This object contains many methods that allow you to interact with other plugins on the bus.

Let's start with the assumption that we've created the below plugin interface:

```
var oMyPlugin = CXBus.registerPlugin("MyPlugin");
```

oMyPlugin.registerCommand

Allows you to register a new command on the bus for other plugins to use.

Example

```
oMyPlugin.registerCommand("test", function(e){

        console.log("'MyPlugin.test' command was called", e)

        e.deferred.resolve();
});
```

Arguments

Name	Type	Description
Command Name	string	The name you want for this command. When other plugins call your command, they must specify the namespace as well. Example: "test" is called on the bus as "MyPlugin.test".
Command Function	function	The command function that is executed when the command is called. This function is provided an **Event Object** that contains metadata and any options passed in.

Event Object

Name	Type	Description
time	number (integer time)	The time the command was called.
commander	string	The name of the plugin that called your command. Example: If your plugin called a command, the value would be "MyPlugin". You can use this information to create plugin-specific logic in your command.
command	string	The name of this command. Example: "MyPlugin.test". This can be useful if you are using the same function for multiple commands and need to identify which commmand was called.
deferred	deferred promise object	When a command is called, a promise is generated. You must resolve this promise in your command without exception. Either execute e.deferred.resolve() or e.deferred.reject(). You may pass values back through these methods. If you pass a value back inside reject() it will be printed in the console as an error log automatically.
data	object	This is the object containing command options passed in when the command was called. If no options were passed, this will default to an empty object.

Returns

Returns true

---

oMyPlugin.registerEvents

Registering events is a formality that allows CXBus to keep a registry of all possible events. You don't need to register events before publishing them, but it's a best practice to always register events.

Example

```
oMyPlugin.registerEvents(["ready", "testEvent"]);
```

Arguments

Name	Type	Description
Event Name Array	array	An array of event names.

Returns

Returns true if at least one value event was included in the array. Returns false if no events are included in the array or no array is passed in.

---

oMyPlugin.subscribe

Subscribes your plugin to an event on the bus with a callback function. When the event is published, the callback function is executed. You can subscribe to any event, even if the event does not exist. This allows for binding events that may come in the future.

Example

```
oMyPlugin.subscribe("WebChat.opened", function(e){

        // e = Event Object. Contains metadata and attached data
        //
        // Example Event Object data:
        //
        // e.time == 1532017560154
        // e.event == "WebChat.opened"
        // e.publisher == "WebChat"
});
```

Arguments

Name	Type	Description
Event Name	string	The name of the event you want to subscribe to. Must include the plugin's namespace.
Callback Function	function	A function to execute when the event is published. An Event Object is passed into this function that gives you access to metadata and attached data.

Event Object

Name	Type	Description
time	number (integer time)	The time the event was published.
event	string	The name of the event, including namespace. That can be useful if you are using the same function to handle multiple events.
publisher	string	The namespace of the plugin that published the event.

Returns

Returns the name of the event back to you if the subscription was successful. Retruns false if you did not specify an event and/or a callback function.

---

oMyPlugin.publish

Publishes an event on the bus under your plugin's namespace.

Example

```
// Publishes the event "MyPlugin.testEvent" with attached data {test: "123"}
oMyPlugin.publish("testEvent", {test: "123"});
```

Arguments

Name	Type	Description
Event Name	string	The name of the event you want to publish. Do not include the plugin namespace.
Attached Data	object	An object of arbitrary properties

Name	Type	Description
		you can attach to your event.

Returns

Always returns true

oMyPlugin.republish

A special method of publishing intended for one-off events like "ready". In some cases, an event will fire only once. If a plugin is loaded at a later time that needs to subscribe to this event, it will never get it because it will never be published again. To solve this problem we have the "republish" method that will automatically republish an event to new subsribers as soon as they subscribe to it.
In Genesys Widgets, every plugin publishes a "ready" event. This event is published using "republish" so that any plugin loaded and/or initialized after can still receive the event.
It is important that you only use "republish" for events that publish once. Using republish multiple times for the same event can cause unwanted behavior.
Genesys Widgets plugins all publish a "ready" event. This is not related to the CXBus plugin interface object's "ready()" method. Calling oMyPlugin.ready() will not publish any events.

Example

```
oMyPlugin.republish("ready", {...});
```

Arguments

Name	Type	Description
Event Name	string	The name of the event you want to have republished. Do not include the plugin namespace.
Attached Data	object	An object of arbitrary properties you can attach to your event.

Returns

Always returns true

oMyPlugin.publishDirect

A slight variation on "publish", this method will only publish an event on the bus if it has subscribers.

The intention of this method is to avoid spamming the logs with events that no plugins are listening to. In particular, if you have an event that publishes frequently or on an interval, "publishDirect" may be used to minimize its impact on logs in the console.

Example

```
oMyPlugin.publishDirect("poll", {...});
```

Arguments

Name	Type	Description
Event Name	string	The name of the event you want to have republished. Do not include the plugin namespace.
Attached Data	object	An object of arbitrary properties you can attach to your event.

Returns

Always returns true

oMyPlugin.command

Have your plugin call a command on the bus.

Example

```
oMyPlugin.command("WebChat.open", {...}).done(function(e){

        // If command succeeds
        // e == any returned data

}).fail(function(e){

        // If command fails
        // e == any returned data

}).always(function(){

        // Always executed
});
```

Arguments

Name	Type	Description
Command Name	string	Name of the command you wish to call
Command Options	string	Optional: An object containing properties the command will use in its execution. Refer to plugin

Name	Type	Description
		references for a list of options available for each command.

Returns

Always returns a promise. You can define done(), fail(), or always() callbacks for every command.

oMyPlugin.before

Introduced: 9.0.003.03

Allows you to interrupt a registered command on the bus with your own "before" function. You may modify the command options before they're passed to the command, you may trigger some action before the command is executed, or you can cancel the command before it executes.
You may specify more than one "before" function for a command. If you do, they will be executed in a chain where the output of the previous function becomes the input for the next function. You cannot remove "before" functions once they have been added.

Example

```
oMyPlugin.before("WebChat.open", function(oData){

    // oData == the options passed into the command call
    // e.g. if this command is called:   oMyPlugin.command("WebChat.open", {form: {firstname:
"Mike"}});
    // then oData will == {form: {firstname: "Mike"}}

    // You must return oData back, or an empty object {} for execution to continue.
    // If you return false|undefined|null or don't return anything, execution of the command
will be stopped
    return oData;
});
```

Arguments

Name	Type	Description
Command Name	string	Name of the function you want to interrupt with your "before" function
"before" Function	function	A function that accepts command options (oData in above example). If you want the command to continue executing, you must return the oData object. If you want to cancel the command, return **false** or **undefined** or don't return

Name	Type	Description
		anything. You may modify the contents of oData before it is sent to the command. This allows you to override command options or add on dynamic options depending on external conditions.

Returns

Returns true when you pass a properly formatted command name (e.g. "PluginName.commandName"). Returns false when you pass an inproperly formatted command name.

oMyPlugin.registry

Returns the CXBus Registry lookup table.

Example

```
oMyPlugin.registry();
```

Arguments

No Arguments

Returns

Returns the internal CXBus registry that tracks all plugins, their commands, and their events.
Registry Structure Example:

```
{
        "Plugin1": {

                commands: ["command1", "command2"],
                events: ["event1", "event2"]
        },

        "Plugin2": {

                commands: ["command1", "command2"],
                events: ["event1", "event2"]
        }
}
```

oMyPlugin.subscribers

Returns a list of events and their subscribers.

Example

```
oMyPlugin.subscribers();
```

Arguments

No Arguments

Returns

Returns an object identifying Example of WebChatService's subscribers:

```
// Format  {"eventname": ["subscriber1", "subscriber2"]}

{

"WebChatService.agentConnected":["CoBrowse","CallUs","ChatDeflection","WebChat","GWE"],
        "WebChatService.agentDisconnected":["CoBrowse","CallUs","WebChat"],
        "WebChatService.ready":["ChatDeflection"],
        "WebChatService.started":["ChatDeflection","WebChat","GWE"],
        "WebChatService.restored":["ChatDeflection","WebChat"],
        "WebChatService.clientDisconnected":["ChatDeflection"],
        "WebChatService.clientConnected":["ChatDeflection","GWE"],
        "WebChatService.messageReceived":["ChatDeflection","WebChat"],
        "WebChatService.error":["WebChat","GWE"],
        "WebChatService.restoreTimeout":["WebChat"],
        "WebChatService.restoreFailed":["WebChat"],
        "WebChatService.ended":["WebChat","GWE"],
        "WebChatService.agentTypingStarted":["WebChat"],
        "WebChatService.agentTypingStopped":["WebChat"],
        "WebChatService.restoredOffline":["WebChat"],
        "WebChatService.chatServerWentOffline":["WebChat"],
        "WebChatService.chatServerBackOnline":["WebChat"],
        "WebChatService.disconnected":["WebChat","GWE"],
        "WebChatService.reconnected":["WebChat"]
}
```

oMyPlugin.namespace

Returns your plugin's namespace.

Example

```
oMyPlugin.namespace();
```

Arguments

No Arguments

Returns

Returns your plugin's namespace. If your plugin's namespace is "MyPlugin", it will return "MyPlugin".

---

oMyPlugin.ready

Marks your plugin as ready to have its commands called. This method is required to be called for all plugins. You should call this method after all your commands are registered, initialization code is finished, and and configuring has completed. Failure to call this method will result in your commands being unexecutable.

Example

```
oMyPlugin.ready();
```

Arguments

No arguments

Returns

Returns nothing

Genesys Widgets Extensions

Genesys Widgets allows 3rd parties to create their own plugins/widgets to extend the default package. Extensions are an easy way to define your own while utilizing the same resources as core Genesys Widgets.

Defining Extensions

Extensions are defined at runtime before Genesys Widgets load. You can define them inline or include extensions in separate files, either grouped or separated.

Important

Define/include your extensions **after** your Genesys Widgets configuration object but **before** you include the Genesys Widgets JavaScript package

```
<script>

if(!window._genesys.widgets.extensions){

    window._genesys.widgets.extensions = {};
}

window._genesys.widgets.extensions["TestExtension"] = function($, CXBus, Common){

    var oTestExtension = CXBus.registerPlugin("TestExtension");

    oTestExtension.subscribe("WebChat.opened", function(e){});

    oTestExtension.republish("ready"); // Publishes "TestExtension.ready"

    oTestExtension.command("WebChat.open").done(function(e){

          // Handle success return state

    }).fail(function(e){

          // Handle failure return state
    });

    oTestExtension.registerCommand("demo", function(e){

          // Command execution here

          e.deferred.resolve(); // or e.deferred.reject(); if the command cannot complete
    });

    oTestExtension.ready();
};
```

```
</script>
```

Make sure that the "extensions" object exists and always include this at the top of your extension definition.

```
if(!window._genesys.widgets.extensions){

        window._genesys.widgets.extensions = {};
}
```

Create a new named property inside the "extensions" object and define it as a function. When Genesys Widgets initializes it will step through each extension and invoke each function, initializing them. Genesys Widgets will share resources as arguments. These include: jQuery, CXBus, and Common (common UI utilities).

```
window._genesys.widgets.extensions["TestExtension"] = function($, CXBus, Common){
```

Creating a new CXBus plugin

Inside the extension function is where you create a new CXBus plugin. You can use this CXBus plugin to interface with other Genesys Widgets. You can add your own UI controller logic in here or simply use the extension to connect an existing UI controller to the bus (for example, share its API over the bus and coordinate actions with events).

Registering a new plugin on the bus creates a new, unique namespace for all your events and commands. In this example, the namespace "cx.plugin.TestExtension" is created:

```
var oTestExtension = CXBus.registerPlugin("TestExtension");
```

Important

When referring to other namespaces, like "cx.plugin.TestExtension", it is not necessary to include the "cx.plugin." prefix. It is optional and implied. You can subscribe to events or call commands using the full or truncated namespace.

Use Cases

Extensions are like any other Genesys Widget. You can publish, subscribe, call commands, or register your own commands on the bus. You can interface with other widgets on the bus for more complex interactions. The following examples demonstrate how you can make extensions work for you.

Example: subscribing to an event.

```
oTestExtension.subscribe("WebChat.opened", function(e){});
```

Example: publishing the "ready" event.

Publishes the event "TestExtension.ready" on the bus. The "ready" event is a standard event for all widgets that is published after the widget has initialized and registered commands on the bus. This is an indicator to other widgets that your widget is ready to interact with others.

The use of republish() here allows for other widgets to load asynchronously and still be notified when others are ready. Republish() will automatically publish the event again **privately** to a new widget that subscribes to it. This allows a one-time event (a state that does not change from its initial state), like "ready" to propagate asynchronously.

Use publish() for all other events.

```
oTestExtension.republish("ready", {arbitrary data to include});
```

Example: publishing a typical event.

You can publish any event at any time and include any data along with it.

Our recommended naming convention is to pair events with commands. e.g. the "open" command publishes "opened" when it is successful. The event should be a past-tense version of the command name.

For command pairs that toggle a state, such as "open" and "close", the names should use common antonyms. Widgets uses the Open/Close convention for showing and hiding widgets.

```
oTestExtension.publish("opened", {arbitrary data to include});
```

Some standard common events used by Widgets (as necessary):

- ready - online - offline - opened - closed - enabled - disabled

Other events are specific to each Widget's function, such as:

- WebChat.messageAdded - WebChat.minimized

Example: calling a command.

Commands are deferred functions. You must handle their return states asynchronously.

```
oTestExtension.command("WebChat.open", {any options required}).done(function(e){

    // Handle success return state
    // "e", the event object, is a standard CXBus format
    // Any return data will be available under e.data

}).fail(function(e){

    // Handle failure return state
    // "e", the event object, may contain an error message, warning, or AJAX response object
});
```

Example: Registering a command.

Creates a new command under your namespace that you or other widgets can call.

"e", the event object, is a standard CXBus format

- e.data = options passed into command when being called.
- e.commander = the namespace of the widget that called this command. (can be used to restrict access)
- e.command = the name of the command being called.
- e.time = timestamp when the command was called.
- e.deferred = the deferred promise created for this command call. You MUST always resolve or reject this promise using e.deferred.resolve() or e.deferred.reject(). You may pass any arbitrary data into either resolution state.

```
oTestExtension.registerCommand("demo", function(e){

    // Command execution here

     e.deferred.resolve();
});
```

Example: Using the 'before()' method

Allows you to set up an interrupt that is executed before a command every time that command is called. With this feature you can link execution of a command with other logic, modify command options before they're used, or cancel execution of a command.

You can specify multiple "before" functions for a single command. They will be executed in order with the output of one providing the input to the next. If one of the functions does not return an object, execution will stop and the command will be cancelled.

```
oTestExtension.before("WebChat.open", function(oData){

    // oData == the options passed into the command call
    // e.g. if this command is called:   oMyPlugin.command("WebChat.open", {form: {firstname:
"Mike"}});
    // then oData will == {form: {firstname: "Mike"}}

    // You must return oData back, or an empty object {} for execution to continue.
    // If you return false|undefined|null, execution of the command will be stopped
    return oData;
});
```