Widgets Developer Resources

Genesys Widgets Extensions

5/1/2026

Contents

- Developer

Learn how to create your own plugins and widgets.

Related documentation:

-

Overview

Genesys Widgets allows you to create your own plugins and widgets. These extensions are an easy way to define your own functionality, while using the same resources as the core Genesys Widgets.

Defining extensions

Extensions are defined at runtime before Genesys Widgets loads. 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.

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 the 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 an event

Publishes the event "TestExtension.ready" on the bus.

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

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.
- 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
});
```

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;
});
```